Universidad Autónoma de Madrid
Escuela politécnica superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

ESTUDIO DEL ESTADO DEL ARTE DE SISTEMAS DE ALMACENAMIENTO DISTRIBUIDO PARA EL ALMACENAMIENTO DE DATOS DE RED

Autor: Jorge Cifuentes Fernández
Tutor: Rafael Leira Osuna
Ponente: Iván González Martínez

JUNIO 2018

ESTUDIO DEL ESTADO DEL ARTE DE SISTEMAS DE ALMACENAMIENTO DISTRIBUIDO PARA EL ALMACENAMIENTO DE DATOS DE RED

Autor: Jorge Cifuentes Fernández
Tutor: Rafael Leira Osuna
Ponente: Iván González Martínez

High Performance Computing and Networking
Dpto. de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
JUNIO 2018

Resumen

Resumen

Las redes de gran escala como Internet están en constante crecimiento, una expansión en parte propiciada por el auge de servicios con un alto uso de ancho de banda, como *streaming* de vídeo. Este desarrollo, junto a ataques contra la seguridad de la red más frecuentes y sofisticados, ha favorecido que el análisis del tráfico que fluye por las redes sea aún más importante para cualquier empresa o institución. Y es que, siendo éstas partícipes de una sociedad global cada vez más interconectada y con una tasa de penetración de la tecnología en los hogares más alta cada año, necesitan estar protegidas ante accesos no autorizados a su infraestructura de red y, por tanto, a sus datos. La monitorización también es vital para detectar infraestructura de red defectuosa o congestionada y así poder realizar un mantenimiento preventivo. Como consecuencia de esto, se abre un complejo escenario donde son necesarios sistemas capaces de, por un lado, guardar la información del tráfico que atraviesa la red de manera segura y por otro, proporcionar un acceso sencillo y rápido a todo este tráfico de datos.

Esta problemática necesita de grandes cantidades de almacenaje, y por lo tanto de muchos discos muy potentes para almacenar estos resultados. Las soluciones orientadas al Big Data son los productos adecuados para esta función y, dentro de estas, hemos estudiado los sistemas de almacenamiento distribuidos. Estos sistemas se basan en el paradigma de repartir la carga de trabajo entre los nodos de una red de ordenadores, buscando mover la computación a donde estén los datos. Cuando se habla de este tipo de software, vienen a la mente Apache Hadoop y su sistema de archivos distribuido HDFS, pero hay un buen número alternativas. En este trabajo de fin de grado analizaremos este último y otros cuatro sistemas de ficheros distribuidos. Primero, hemos comprobado su rendimiento como sistemas de almacenamiento aislado, y después como proveedores de almacenaje a un sistema de procesamiento distribuido. Nuestro objetivo es encontrar las diferencias, tanto teóricas como prácticas, de las soluciones disponibles en el mercado para este complejo problema, y realizar así un análisis del estado del arte en el que hemos encontrado diferencias interesantes.

Palabras Clave

Big Data, sistemas de archivos distribuidos, computación distribuida, análisis de red

Abstract

Big scale networks such as the Internet are constantly growing, an expansion partly stimulated by the upswing of services with a bandwidth-intensive use like video streaming. This development, alongside more frequent and sophisticated network attacks, has favoured network traffic analysis to be even more relevant to any company or institution. As key parts in a increasingly interconnected global society and with a penetration rate of technology in households higher every year, they need to be well protected against unauthorized accesses to their infrastructure and, therefore, their data. Monitoring is vital as well to detect underperforming or congested infrastructure so a preventive maintenance can take place. Thus, a complex setting appears where we need systems capable of both storing all this network traffic safely and to provide an easy and fast way to access them.

This problematic needs a huge storage volume, and a lot of powerful disks to keep the data. Big Data solutions are the adequate products to fill this gap, and more specifically we researched about distributed file systems. These systems are based on the *split the data and the computation* in a computers network paradigm, aiming to move the computation near the data. When we talk about this kind of software, one powerful player is Apache Hadoop and its distributed file system HDFS, but there is a lot of other choices. In this essay we have analyzed HDFS and other four of its counterparts. First, we have tested their performance solely as storage systems, and subsequently as data providers to a distributed computing system. Our goal is to find the differences, whether in the theory or in the analysis, and complete a research on the distributed file systems state of art, where we have found interesting differences.

Key words

Big Data, distributed file systems, distributed computing, network analysis

Agradecimientos

A Rafa, mi tutor, por su gran capacidad de transmitir parte de sus amplios conocimientos y de plantear nuevos retos.

A todos mis colegas de grado en estos años y a todas las personas que he conocido y con los que he compartido innumerables horas de biblioteca y, especialmente, laboratorios. Se me han pasado muy rápido estos cinco años.

A mi familia.

Índice general

Índice de Figuras

Índice de Tablas

Glosario

AFR Automatic File Replication. 9

API Application Program Interface. 6, 7

CFQ Completely Fair Queuing. A.I

CRUSH Controlled Replication Under Scalable Hashing. 12

DFS Distributed File System. 1, 2, 6, 7, 16, 31–33, 36, 37, 41, 42

DHT Distributed Hash Table. 9, A.II

DPC Data Processing Center. 8

FIFO First In First Out. A.I

fio flexible input-output. 15, 16

FUSE Filesystem in Userspace. 6, 7, 9, 10, 16, 29, 32, 40, 41

HDD Hard Disk Drive. 13, 15, 18, 31

HDFS Hadoop Distributed File System. 1, 7, 8, 14–16, 18–35, 37, 38, 40, 41, A.I

HTTP Hypertext Transfer Protocol. 35

JAR Java ARchive. 36, 37

JNI Java Native Interface. 34, 40, 41

JVM Java Virtual Machine. 34, 35, 37, 38, A.I, A.II

LNet Lustre Networking. 13, A.III

MDS MetaData Server. 10, 11, 13, 14

MDT MetaData Target. 10, 11, 13

MGS Management Server. 10, 11, 13

MGT Management Target. 13

MSD Metadata Server Daemon. 11

OSD Object Storage Daemon. 11–13

OSS Object Storage Server. 10–14

OST Object Storage Target. 10–14

POSIX Portable Operating System Interface. 15

QJM Quorum Journal Manager. 7

RADOS Reliable Autonomic Distributed Object Store. 11, 12

RAID Redundant Array of Independent Disks. 13

SPOF Single Point of Failure. 6, 7, 11

SSD Solid State Drive. 12, 13, 18, 32

TCP Transmission Control Protocol. 24, 31

TSP Trusted Storage Pool. 9

VFS Virtual File System. 6, 7

1 Introducción

La complejidad creciente de las redes de gran escala, con una infraestructura de red cada vez más rápida para dar respuesta a servicios con un alto uso de ancho de banda, como vídeo por streaming, plantea nuevos retos de cara al manejo de dicho tráfico de red. Además, una tasa de uso de Internet [1] en continua expansión, implica dificultades cada vez mayores de cara al análisis efectivo de las redes de ordenadores.

El estudio del tráfico de red es un punto importante que puede ser usado tanto para optimizar el servicio, como para detectar posibles brechas de seguridad y ataques maliciosos [2]. En este escenario de grandes cantidades de información siendo transmitida por la red, necesitamos un sistema que nos dé la capacidad de guardar estos datos así como de acceder de manera rápida a ellos. Aquí es donde entran en juego las soluciones orientadas al Big Data, capaces de procesar grandes cantidades de datos de manera veloz.

En general, estos sistemas de Big Data, están conformados por sistemas de almacenamiento y sistemas de procesamiento. Su operativa se basa en la idea de distribuir entre varias máquinas participantes de un cluster tanto los datos como la ejecución de programas sobre esos datos. El concepto clave es mover el procesamiento a donde estén los datos, y no mover los datos a donde esté el programa ejecutándose. De esta manera, es posible escalar el rendimiento usando *off-the-shelf hardware* o hardware de estantería[3, cap. 1], es decir, máquinas con componentes que pueden ser comprado en cualquier establecimiento. Así pues, buscan dar al usuario un modo seguro, resiliente a fallos y rápido de acceder y tratar los datos.

Tal vez el más conocido sea el framework Apache Hadoop, compuesto por su sistema de almacenamiento distribuido Hadoop Distributed File System (HDFS) y un sistema de procesamiento distribuido llamado YARN, pensados para almacenar en varias máquinas terabytes de datos y de proporcionar un acceso eficiente a ellos. Por ejemplo Facebook, entre muchos otros usuarios de este framework[I], opera un cluster Hadoop con 1100 nodos, 8800 cores y 12 petabytes de almacenamiento. También Spotify[II] mantiene un cluster con 70 terabytes de RAM y 65 petabytes distribuidos entre más de 1600 máquinas. Vistas las capacidades de este software en concreto, hay que destacar que existe gran cantidad de sistemas de ficheros distribuidos aparte de HDFS y por ellos buscaremos alternativas, y presentaremos un estudio comparativo.

Estamos por tanto ante el problema de almacenar ingentes cantidades de archivos, específicamente tráfico de red, de manera rápida y eficiente, que consiga mantener el ritmo de redes con tránsito intenso de datos. Además, también esta la problemática de acceder velozmente a estos datos una vez estén en reposo. Por ello, el objetivo final de este trabajo de fin de grado es usar y comparar el rendimiento de varios de estos Distributed File System (DFS), primero como sistema de almacenamiento de tráfico de red, y luego como parte de un sistema de procesamiento que analizará dicho tráfico. Para ello, se instalarán estos sistemas de archivos en un pequeño cluster y se probará sobre él. En primer lugar, se realizarán pruebas de rendimiento únicamente buscando evaluar las capacidades como sistema de almacenamiento, en varios casos de uso concretos. A continuación, se desarrollará un programa que será ejecutado de manera distribuida en nuestro cluster. Este software analizará paquetes de red presentes previamente en el cluster, con el fin de comprobar cuán rápidos son en procesar una cantidad importante de tráfico.

[I] Propulsado por Apache Hadoop: `https://wiki.apache.org/hadoop/PoweredBy`

[II] `slideshare.net/AdamKawa/hadoop-adventures-at-spotify-strata-conference-hadoop-world-2013`

1.1. Objetivos

Durante este trabajo de fin de grado analizaremos diferentes soluciones al problema planteado de almacenamiento distribuido de grandes cantidades de datos, específicamente de archivos que contengan el tráfico de la red, así como también sobre el procesamiento distribuido de dichos datos, una vez hayan sido almacenados y estén en reposo.

En cuanto a lo relacionado con los sistemas de almacenamiento, un objetivo de aprendizaje es conocer y entender su arquitectura básica, y comprender el funcionamiento del engranaje formado por todas las partes de cada sistema de archivos analizado. Después, buscaremos obtener el rendimiento de estos sistema de ficheros respecto a varios parámetros como tamaños de archivo o tamaño de bloque. Para el análisis de resultados haremos especial hincapié en señalar los comportamiento tanto beneficiosos como perjudiciales para nuestro problema de almacenamiento de tráfico de red.

Por otro lado, también será un objetivo conocer los conceptos de computación distribuida y su relación con el almacenamiento distribuido. Aplicaremos esta teoría de manera práctica desarrollando un software con Apache Hadoop con el fin de medir el rendimiento de nuestros DFS. Estos sistema de ficheros formarán en este punto parte de un framework más grande dedicado al procesamiento distribuido del tráfico de red. El propósito final de este programa será obtener ciertas estadísticas analizando todos los datos de un archivo con tráfico de red de manera distribuida, así como analizarlos y entender los resultados y su relación con el problema planteado.

Relacionado con esto último, otro objetivo que también merece ser nombrado es el aprendizaje del uso de la herramienta flow_process, amablemente cedida por el tutor de este trabajo. Este programa realiza un procesamiento bastante completo del tráfico de red, lo que nos sirve perfectamente como herramienta de análisis dentro de nuestro software de computación distribuida.

En definitiva, el objetivo de este trabajo es conocer, usar y probar las diferentes soluciones software al problema planteado, y analizar según los resultados obtenidos cuales se ajustan mejor a nuestro escenario de uso.

1.2. Fases de realización

En primer lugar, fue necesario un estudio teórico, indispensable para todo el trabajo posterior, de los ya nombrados sistemas de almacenamiento y computación distribuida de cara al objetivo de aprendizaje de los mismos. Teniendo clara la teoría general, se pasó a estudiar la arquitectura de cada uno de estos sistema de archivos distribuidos elegidos.

Una vez con el conocimiento teórico de cada DFS en mente, la metodología seguida fue instalar primeramente el software y conseguir que funcionase de la manera correcta. Con el software ejecutándose, también se indagó sobre varias posibles optimizaciones, que dan lugar al apéndice A. Después de cada instalación y puesta a punto, se ejecutaron las extensas pruebas de rendimiento.

Buscando unos resultados confiables, las pruebas se ejecutaron múltiples veces por cada valor de cada parámetro usado, con la meta de encontrar valores reales y reproducibles. Para automatizar la repetición de las pruebas, se desarrolló un script de bash que se encargara de ellos, con la mínima intervención humana.

Recopiladas todas las pruebas en formato `csv`, y siguiendo con ese espíritu de automatizar para facilitar las tareas repetitivas, se creó un script de bash que, junto a un script de Octave (el equivalente libre de MATLAB), tratara estos archivos, realizando las medias necesarias, y creara todas las gráficas de manera rápida y fiable para poder proceder a cumplir el objetivo de su análisis y estudio.

En lo respectivo al capítulo 4, una vez comprendido y ganado soltura con el programa `flow_process`, fue necesario un concienzudo estudio teórico del uso en el lenguaje de progra-

mación Java del framework Apache Hadoop, de todos sus componentes y clases, para así poder programar un producto ejecutable con él. Esto dio resultado a un ejecutable después usado como medida de rendimiento, y a diversas gráficas que de nuevo tuvieron que ser estudiadas y enlazadas con el conocimiento teórico.

1.3. Estructura del documento

En el segundo capítulo se hace primero una introducción a los sistemas de archivos distribuidos de manera genérica, nombrando sus partes más importantes. Después, en las subsiguientes secciones se describen los cinco sistemas de archivos distribuidos que hemos usado en nuestras pruebas y que consideramos que son los más representativos del mercado. En concreto, nos hemos fijado en la arquitectura general, listando y definiendo las partes que los componen, explicando como funcionan en conjunto, y del flujo de datos, es decir, del funcionamiento interno del software al ejecutar operaciones sobre archivos.

Después, en el capítulo tercero, se definen y explican las pruebas realizadas, justificando su sentido y utilidad, para seguidamente presentar en formato gráfica los resultados de estas pruebas, así como una pequeña reflexión sobre los resultados, especialmente comparándola con las capacidades teóricas de cada sistema de ficheros en el capítulo segundo y enfocándolo a nuestro problema. Este capítulo se cierra con un resumen detallado de los resultados de las pruebas.

Pasamos al cuarto capítulo, que versa sobre el software desarrollado para hacer pruebas en un entorno algo más real, de procesamiento distribuido, que las del capítulo 3. Se introduce el concepto de MapReduce, para después explicar su relación con el framework de nuestra elección y finalmente aunar todo el conocimiento explicando la implementación del software desarrollado. Más adelante, se muestran gráficas con los resultados obtenidos en varios aspectos, seguido de una pequeña conclusión sobre los resultados, de nuevo encuadrándolo en nuestro problema de almacenamiento de tráfico de red.

Este trabajo se cierra con el quinto capítulo, en el que se expondrá una conclusión global, y también se darán diferentes puntos por donde se podría abordar un trabajo futuro. Al final del trabajo están presentes la bibliografía y dos apéndices. El primero detalla las optimizaciones sobre los sistemas de archivos que usamos en el capítulo 3, y en el segundo se expone el código desarrollado en el capítulo 4 para tener un acceso rápido a el.

2 Estado del arte

2.1. Introducción a los sistemas distribuidos

Un sistema de archivos se considera distribuido cuando sus contenidos están repartidos en diferentes máquinas o nodos, funcionando bajo un espacio de nombres común. El acceso por parte de las máquinas cliente es transparente y a través de la red, ya que éste visualiza y accede a los datos como si fuera un sistema de archivos local, sin conocer los detalles de la ubicación de los archivos. Es el propio sistema de ficheros el que, por detrás, gestiona la comunicación de los nodos y el almacenamiento de los datos.

En general, buscan ser capaces de funcionar en sistema heterogéneos (diferentes sistemas operativos y hardware), ser fácilmente escalables (mantener un rendimiento creciente y lineal respecto al número de nodos), ser consistentes (todos los procesos ven los archivos en el mismo estado) y ser capaces de recuperarse de fallos.

A nivel de software, estos sistemas de archivos normalmente constan de varios procesos (a veces también llamados servidores o demonios, dependiendo de la terminología de cada uno), que operan de manera coordinada cumpliendo diferentes funciones. Suele haber un proceso de gestión del sistema, otro de gestión de metadatos y también de gestión de archivos. Lo idóneo en este tipo de sistemas es tener procesos descentralizados (no queremos un punto de acceso único que cree un sistema frágil) y *cluster-aware*, donde los procesos se conozcan entre sí y puedan establecer comunicación sin necesitar un gestor central.

2.1.1. Distribución y replicación

Vamos a manejar dos conceptos importantes: distribución y replicación. La distribución se refiere al hecho de que los archivos estén repartidos en diferentes discos en diferentes máquinas. Lo más habitual no es escribir los archivos enteros en los discos, sino partirlos en bloques (*striping* o particionado), que serán distribuidos. La idea de operar así es optimizar al máximo el uso de los discos: escribir un archivo de 1GB entero en un disco con una velocidad de, por ejemplo, 100MB/s costaría 10 segundos. Si partimos el archivo en 10 bloques de 100MB y escribimos estos bloques en 10 discos en paralelo, tendríamos una velocidad ideal de 1 segundo. Por lo tanto, y de manera teórica, el tener más discos con los que poder hacer I/O de manera paralela incrementa el rendimiento en un factor igual a ese número de discos. De manera práctica en las pruebas comprobaremos que el rendimiento no escala igual que el número de discos, por el propio *overhead* ocasionado por el software. Por lo tanto, este software tiene que ser capaz tanto de hacer este reparto cuando el cliente escriba un archivo, como de localizar los bloques desperdigados cuando quiera hacer una lectura.

El otro concepto clave es la replicación: mientras que la distribución y el particionado buscan incrementar el rendimiento, la replicación busca mantener la tolerancia frente a fallos. En un sistema distribuido, sin replicación, un fallo en un nodo o un reinicio por mantenimiento supondría que cualquier archivo con algún trozo almacenado en él quedara inutilizado. Con la replicación, los bloques de datos se almacenan repetidos un número de veces conocido como factor de replicación. Estos bloques estarán en diferentes discos, racks o incluso salas de servidores, dependiendo de la configuración y flexibilidad del software. Lógicamente, esto tiene varias implicaciones con respecto la lectura y escritura. El rendimiento de esta última se verá perjudicado, ya que por cada bloque de datos que el cliente quiera escribir, el software tendrá que escribir varias veces. Además, el cliente solo tendrá disponible el tamaño total de los discos del

sistema dividido entre el factor de replicación. En cuanto al rendimiento en la lectura, este se verá potencialmente mejorado, especialmente en sistemas con una carga de uso considerable. Esto es debido a que si el disco donde se almacena un bloque que el sistema necesita está ya ocupado realizando operaciones de I/O, irá a buscarlo al siguiente disco en proximidad donde esté almacenado, evitando quedar bloqueado a la espera.

En resumen, aumentar el factor de replicación supone una perdida palpable de rendimiento en escritura y capacidad de almacenamiento, pero una ganancia en tolerancia a fallos y potencialmente en rendimiento en lectura. Por lo tanto, lo recomendable y más estandarizado para mantener el compromiso entre rendimiento y disponibilidad es un factor de replicación de 2 o 3.

Alta disponibilidad Aparte de la distribución y replicación como concepto para mejorar el rendimiento y la tolerancia frente a fallos, otro criterio para valorar los sistemas de archivos distribuidos es la alta disponibilidad (*high availability*). En nuestro caso en concreto, hay dos puntos críticos: Single Point of Failure (SPOF) y recuperación ante fallos. La eliminación de los SPOF viene de la mano de la descentralización, ya que puntos centralizados de acceso al sistema complejo (por ejemplo, un nodo central de metadatos) pueden provocar que el sistema quede inaccesible si fallan, además de crear un posible efecto de cuello de botella. Asimismo, para gestionar dicha descentralización, en los DFS se suele aplicar el concepto de *cluster-aware*, es decir, que los diferentes servicios en los nodos conozcan la topología del cluster para poder intercomunicarse sin necesidad de acudir a un nodo central.

Relativo a la recuperación ante fallos, algunos sistemas de archivos tanto distribuidos como no distribuidos implementan el Journaling. Funciona como un registro donde se lleva la cuenta de las transacciones atómicas efectuadas por los clientes, de manera que funcione como punto de back-up desde el que partir en caso de que el sistema necesite restauración después de un error. El sistema escribe el Journal a disco antes de escribir los datos modificados, de modo que si el sistema falla el software recorre el log para devolver el sistema a un estado consistente. En los DFS sirve para mantener la consistencia de datos entre los diferentes servicios y máquinas, pero puede dañar el rendimiento cuando éste usa dispositivos de almacenamiento subóptimos [4].

2.1.2. Acceso a los datos

Por último, cabe hablar de las maneras que tiene el cliente de acceder al sistema de archivos. En este trabajo veremos tres: virtual, mediante Filesystem in Userspace (FUSE) y con módulo de kernel, también conocido como módulo nativo.

Un sistema de archivos virtual tiene una capa de abstracción más sobre el sistema de archivos del sistema operativo y se accede a los datos a través de llamadas propias del software del sistema distribuido. Por el contrario, FUSE y el uso de módulos de kernel se basan en acceder al contenido como si fuera una carpeta más de la máquina cliente. Éste puede hacer las mismas operaciones que con cualquier otra carpeta de su sistema operativo, y es el software el que por detrás gestiona la distribución, replicación y demás operaciones. La diferencia entre uno y otro es que, mientras que el módulo de kernel se ejecuta en espacio de núcleo, FUSE se ejecuta en espacio de usuario llamando a la API `libfuse`, que es la que gestiona la comunicación con el kernel.

En la figura 2.2 se puede ver el flujo de acceso mediante FUSE. Primero, la aplicación de usuario, que usa la librería `glibc`, manda una petición a Virtual File System (VFS). VFS es una Application Program Interface (API) que abstrae el acceso de las aplicaciones a los ficheros, por lo que el sistema de archivos subyacente (ext3, FAT, etc) implementa ciertas funciones comunes dadas por VFS como `open()`, `create()` y demás. En el paso 2, VFS se comunica con un archivo especial, `/dev/fuse`, que es cargado por el módulo de kernel de FUSE y sirve para establecer conexión a través de una API con una aplicación de usuario: el sistema de archivos FUSE ejecutándose como proceso en el espacio de usuario (paso 3). Para que dicha aplicación funcione como sistema de archivos FUSE-compatible, debe haber implementado esta API dada por `libfuse`. Una vez que la aplicación devuelve los datos (la lista de ficheros de la carpeta `myfuse` en este caso) hace el camino de vuelta hacia la aplicación del usuario. Cada vez que se cruza la línea negra horizontal, se realiza un cambio de contexto.

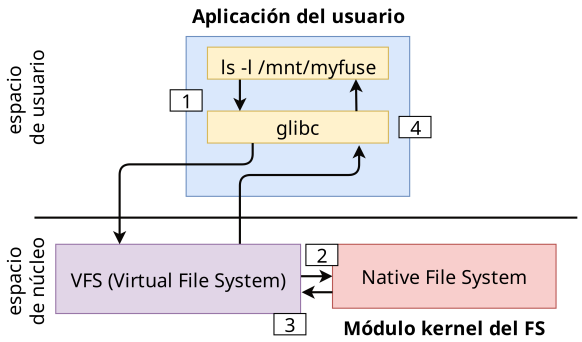


Figura 2.1: Esquema de acceso por módulo nativo para Sistemas de Archivos Linux

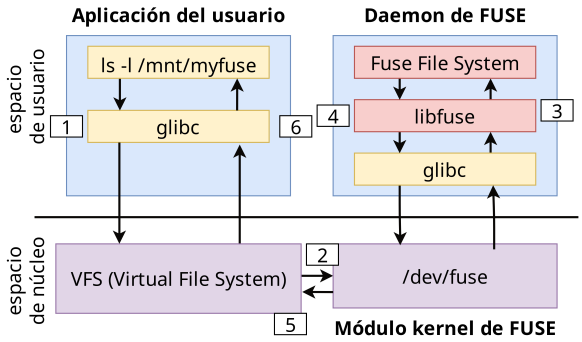


Figura 2.2: Esquema de acceso FUSE para Sistemas de Archivos Linux

En la figura 2.1 en cambio vemos que, si los desarrolladores de nuestro DFS han implementado un módulo de kernel, VFS puede conectarse con él y devolver los datos a la aplicación usuario. El primer caso es más sencillo de llevar a cabo para los desarrolladores que crear un módulo de kernel dado que solo necesitan implementar la API dada por FUSE y dejar que este se encargue de la comunicación con el núcleo. Sin embargo, estos cambios adicionales de contexto, de espacio de usuario a espacio de kernel, pueden lastrar el rendimiento [5]. FUSE también parece tener un rendimiento deficiente en creaciones de archivos y lecturas [6], lo cual puede ser decisivo en sistemas que escriben los datos una vez y lo procesan múltiples veces.

Ahora que hemos relatado de manera general algunos conceptos clave, vamos a pasar a explicar el funcionamiento de los diferentes sistemas de archivos distribuidos que hemos probado. En concreto, en este estudio vamos a comparar cinco ampliamente usados: HDFS 2.7.3, GlusterFS 3.10.10, BeeGFS 6.17, CephFS 10.2.10 y Lustre 2.10.3.

2.2. HDFS

2.2.1. Arquitectura

HDFS está orientado al procesamiento de grandes cantidades de datos en lotes, ya que asegura poder mantener una buen rendimiento de manera continua. En cambio, no está recomendado en entornos con alto nivel de interacción con el usuario, debido a que no dispone de una baja latencia [3]. Sigue la filosofía de escribe una vez-lee múltiples veces, por lo que está diseñado buscando un mayor rendimiento de lectura. También sigue el concepto de mover la computación a donde estén los datos, antes que mover los datos desde donde estén y luego operar sobre ellos.

HDFS mantiene una arquitectura de maestro/esclavo. Se compone de un nodo **NameNode** (maestro) y un número indeterminado de **DataNodes** (esclavos). El NameNode mantiene el espacio de nombres (con ordenación jerárquica tradicional) y todos los metadatos del sistema de archivos, log de ediciones (escritura, apertura y cambio de nombre), y mapea cada identificador de bloque de datos con el identificador del DataNode donde está alojado. Por su parte el DataNode se ocupa de las peticiones de lectura y escritura de los clientes y de crear, replicar, y borrar los bloques de datos.

Parece claro el defecto de esta arquitectura: el NameNode se convierte en un SPOF dado que, si este deja de funcionar, los clientes no sabrían donde alojar o buscar los bloques de datos en la escritura y lectura y el sistema de archivos sería inaccesible. Para evitar esto, en últimas versiones de HDFS se ha introducido el High Availability NameNode, usando Quorum Journal Manager (QJM). QJM se basa en tener dos NameNodes, uno activo y otro en stand-by con el que poder hacer un cambio rápido en caso de fallo del primero. Con QJM, cualquier cambio en el log del NameNode Activo se comunica a varios nodos específicos (JournalNodes). El NameNode en Stand-by monitoriza estos cambios y los aplica a su propio espacio de nombres. De esta manera, si el NameNode principal falla, el secundario tiene todos los datos actualizados.

2.2.2. Distribución, particionado y replicación

Los bloques, también llamados chunks en la terminología de HDFS, ocupan 64MB por defecto (los archivos de menos de ese tamaño no ocupan un bloque entero, lo que evita la fragmentación interna). Los bloques son de un tamaño grande, debido a que HDFS está diseñado para almacenar archivos grandes (típicamente en el rango de los gigabytes o incluso terabytes). Estos archivos pueden ser más grandes que un disco físico. Cada archivo se divide en bloques de tamaño igual, excepto el último.

No guarda relación este tamaño de bloque con el del sistema operativo por debajo. HDFS no crea archivos más grandes que su tamaño de bloque, que el sistema operativo divide en bloques intentando que sean adyacentes, como haría con cualquier archivo para intentar reducir la fragmentación. Este tamaño y el factor de replicación pueden ser configurados a nivel de archivo. Ésta es gestionada enteramente por el NameNode, con el algoritmo *Rack Awareness Algorithm*, que asegura que las replicas no compartan Rack con el bloque original. Se hace en pipeline, como describiremos más adelante.

Como último, hay que comentae algunos elementos interesantes de HDFS para mantener la alta disponibilidad: los heartbeats, el rebalanceo de bloques y la comprobación de la integridad de los datos. Los heartbeats son mensajes que los DataNodes envían periódicamente mensajes al NameNode para confirmar que siguen activos. Si un DN muerto causa que el factor de replicación de algunos bloques de datos baje, el NameNode se encarga de gestionar la replicación adicional. El rebalanceo de bloques es una técnica de redistribución de los datos usada para mantener DataNodes lo más uniformes posibles.

En cuanto a la integridad de los datos, un archivo importante en HDFS es el `FSIMAGE`. Este archivo contiene el espacio de nombres, el mapeado de bloques y las propiedades del sistema. Por tanto, puede ser usado como punto de restauración del estado del sistema.

2.2.3. Flujo de datos

- **Lectura** Cuando el cliente abre un archivo, contacta con el NameNode, que le devuelve las direcciones de los DataNodes donde se almacenan copias de los bloques de ese archivo, ordenados por proximidad. El cliente contacta para cada bloque al DataNode más próximo y los bloques correspondientes se leen de manera paralela.

- **Escritura** Primero se crea el archivo en el espacio de nombres. Los datos se parten en bloques y se llevan a una cola de datos (data queue). Aquí es donde entra en juego el factor de replicación, ya que la escritura se hace en pipeline. Se genera una lista de DataNodes que albergaran los datos de la cola. Cada bloque encolado es enviado a un DataNode, que a su vez lo pasa al siguiente DataNode para que lo escriba, y así sucesivamente hasta que un número de DataNodes igual al factor de replicación tienen una copia del bloque, momento en el que se avisa al cliente del final de la escritura. Por lo tanto la replicación es asíncrona, es decir, cuando el cliente termina su escritura significa que el archivo ya está replicado.

  Este pipeline se hace de manera paralela, para evitar que el tiempo de escritura se multiplique por el factor de replicación, por lo que se divide el bloque de datos en trozos de 4KB, enviando estas piezas en paralelo, incrementando así la concurrencia. El hacerlo así en vez de, por ejemplo, que el cliente escriba directamente a todos los DataNodes tiene ciertas ventajas:
  - El cliente solo envía una vez los datos, y luego estos se reenvían dentro del cluster, lo que quedaría favorecido por el concepto de proximidad, además de mantener una carga de red balanceada. También minimiza costes si el Data Processing Center (DPC) proveedor de nuestro cluster cobra más caro el tráfico externo que en el interno.
  - El cliente tiene una ventana de recepción menor, ya que para él la replicación es totalmente transparente, y solo espera una confirmación de un DataNode.

  Cabe destacar una técnica usada para optimizar la escritura en archivos pequeños desde el

lado cliente, llamada *staging* (o *client-side buffering*). Éste escribe los datos a un archivo temporal local que, cuando está lleno (ocupa el tamaño de un bloque), se escribe en un bloque entero. Esto permite mejorar la velocidad de red y reducir la congestión, ya que no se accede tan frecuentemente al NameNode si los datos son pequeños y ocupan menos de un bloque.

- **Borrado** Cuando un cliente borra un archivo, contacta con el NameNode y éste lo renombra en la carpeta trash, por lo que es recuperable. Pasado un tiempo establecido, el NameNode lo borra del espacio de nombres y libera sus bloques. Dicho tiempo es configurable, en minutos, en la propiedad `fs.trash.interval` del archivo de configuración `core-site.xml`.

2.3. GlusterFS

2.3.1. Arquitectura

Como inicio, hay que explicar que en la terminología de GlusterFS, un brick es una unidad de almacenamiento (un disco), y un conjunto de bricks forman un Volumen, que es toda la capacidad de la que vamos a disponer como clientes. GlusterFS usa el método de acceso FUSE, que ya se comentó en la figura 2.2. El proceso cliente consiste de una pila de traductores, que convierten solicitudes de usuario en solicitudes de almacenamiento. El primero de estos traductores es la propia librería `libfuse`.

En la arquitectura de GlusterFS hay dos servicios básicos:

- El **glusterd** el el demonio de gestión del sistema, es descentralizado y se ejecuta en todos los nodos del `Trusted Storage Pool (TSP)` [7]. La TSP es una red conformada por varias máquinas que proveerán de bricks al sistema.
- El **glusterfsd** es el demonio que se ejecuta en cada brick y maneja las solicitudes de operaciones de datos.

GlusterFS usa el concepto de traductores, siendo los dos más importantes Distributed Hash Table (DHT) y Automatic File Replication (AFR). DHT es el encargado del enrutamiento, tanto de dirigir las escrituras hacia los bricks como de obtener la rutas de éstos para la lectura. AFR por su parte gestiona, cuando la hay, la replicación, replicando un número de veces igual al factor de replicación las solicitudes de lectura o escritura, lanzando un hilo para cada una. Cada uno de estos hilos usa el traductor Protocol Client Translator, que es el que conecta directamente con el `glusterfsd` de cada brick para escribir o leer.

2.3.2. Distribución, particionado y replicación

GlusterFS tiene la peculiaridad de que no se establece un tamaño de bloque fijo, sino que la filosofía es que el archivo se divide en un número de partes múltiplo del número de bricks disponibles, un valor llamado `stripe count` [7]. También hay que dejar constancia del concepto de grupo de repliación: el software crea conjuntos de bricks del tamaño del factor de replicación. Por ejemplo, si dicho factor fuera 3 y tuviésemos 30 bricks, tendríamos 10 grupos, y dentro de cada grupo los 3 discos participantes tendrían el mismo contenido replicado. Así, en este caso, un archivo de 200MB se partiría en 10 (o múltiplo de 10) bloques de 20MB, y cada grupo de replicación tendría los mismos 20MB repetidos tres veces. GlusterFS se puede configurar en varios modos, que son una combinación de las tres principales acciones: distribuir, partir en bloques y replicar. Con mayor importancia cabe comentar:

- **Distribuido y particionado**: Se dividen los archivos en un número igual de partes al `stripe count`. Después, estas partes se distribuyen entre los bricks que conforman cada grupo de striping, un concepto similar a los grupos de replicación pero relativo a la distribución de bloques. En la imagen 2.3 se ve la estructura de este modo, en la que el archivo se divide en 4 bloques. Cada grupo de striping tiene todos los bloques del archivo. Esta estructura sería la ideal en casos con factor de replicación 1.

- **Distribuido, particionado y replicado**: Similar al anterior, se dividen los archivos en bloques, y se distribuyen en los grupos de striping. Dichos grupos conforman a su vez grupos de replicación. En este caso, como se aprecia en la figura 2.4, los 4 bloques se replican con factor 2. Pero esta vez cada grupo de striping, en vez de tener directamente dos discos, está formado por dos grupos de replicación. Por tanto la estructura de GlusterFS es escalonada.

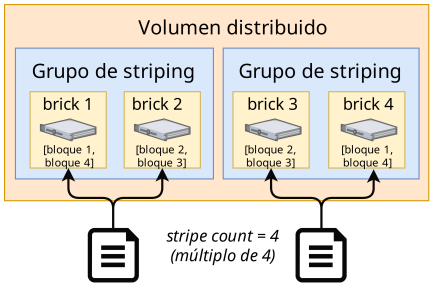


Figura 2.3: Estructura de un volumen de GlusterFS distribuido y particionado

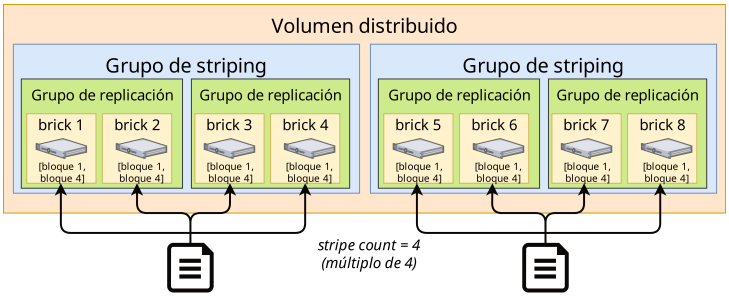


Figura 2.4: Estructura de un volumen de GlusterFS distribuido, particionado y replicado

El hecho de que a cada disco se le asignen bloques del mismo tamaño, puede causar bajadas de rendimiento y discos ociosos en montajes con discos claramente más lentos que otros en el mismo cluster. Para solventar esto, desde la versión 3.6, es posible establecer tamaños de bloque diferentes para discos en concreto.

2.3.3. Flujo de datos

- **Lectura y escritura** Cuando el usuario monta el servicio cliente, este se comunica con el `glusterd` del servidor, que le envía una lista de los bricks del volumen y los traductores disponibles, para que pueda comunicarse directamente con cada brick. Una vez que se emite una solicitud de I/O, esta se envía al módulo FUSE, como ya se nombró antes. Después, se sigue ejecutando la pila de traductores.

- **Borrado** Cuando un archivo se borra, se mueve a una carpeta oculta `.trashcan` de su brick correspondiente, cuyos contenidos puedes sen eliminados por el usuario. Gracias al traductor de basura, el agregado de todos los directorios `.trashcan` se puede acceder desde el cliente.

2.4. BeeGFS

2.4.1. Arquitectura

BeeGFS se compone de varios servidores [8] (en el sentido de servicios o procesos ejecutándose en una máquina):

- El nodo maestro es el **Management Server (MGS)**, que comunica a los demás servicios, pero no influye en las operaciones de I/O. Mantiene una lista de los archivos del sistema de ficheros y su correspondiente información.

- En BeeGFS, se usa el término target para referirse a un medio de almacenamiento como un disco duro. Por tanto, un disco encargado de almacenar metadatos es conocido como **MetaData Target (MDT)**. Cada MDT es usado por un proceso llamado **MetaData Server (MDS)**, encargado de gestionar los metadatos y el espacio de nombres (uno por máquina).

- El **Object Storage Server (OSS)** (uno por máquina) cumple la misión de almacenar los datos. Cada OSS se asocia con uno o varios **Object Storage Target (OST)** (discos físicos). Estos target almacenan dichos objetos o bloques.

2.4.2. Distribución, particionado y replicación

En BeeGFS, el tamaño de bloque define la cantidad de datos que se escribirán en cada MDT o OST antes de pasar al siguiente, teniendo así una distribución de manera circular.

Por otro lado, la replicación es conocida como Buddy Mirroring (mirroring tanto de metadatos como de contenidos). El objetivo de esta técnica es tener nodos que se comuniquen entre ellos para hacer la replicación de bloques, así como que detecten cuando el otro está fuera de servicio. Esto resulta interesante ya que estos nodos pueden estar situados lejanos, diferentes racks, sala de servidores, etc. Si uno de los Buddys falla, se marca como offline y se usa el otro tras un corto periodo de tiempo que asegure que la noticia se propaga a todos los nodos, de manera que se conserve la consistencia.

2.4.3. Flujo de datos

- **Lectura** Para abrir un archivo, el cliente contacta directamente con un MDS, que le reporta los OST donde están los bloques de datos que busca, y el cliente los lee. En caso de haber Buddy Mirroring, los bloques se leen por defecto desde el Buddy primario, pero si está ocupado puede ser el secundario.

- **Escritura** El archivo es partido en bloques y se genera una lista, del tamaño del factor de replicación, de OST que albergarán cada bloque de datos. El cliente contacta directamente con cada OSS y MDS y le escribe cada bloque, rotando de manera circular en la lista. Por ejemplo, en un caso con replicación 3, y con un archivo que es partido en 30 bloques, se escribiría de manera rotativa en el OST1, OST2 y OST3 diez veces.

  Este acceso directo para hacer las operaciones de I/O consigue que se distribuya la carga de trabajo, y evita que haya SPOF. En caso de que haya Buddy Mirroring, el cliente contacta con el Buddy marcado como primario y le escribe su bloque, y este es el que se lo pasa al Buddy Secundario.

- **Borrado** Cuando un archivo es borrado, se mueve a una carpeta `disposal` en el MDT, y sus bloques de datos a carpetas de igual nombre en cada OST. En el momento en que todos los procesos que lo pudieran estar usando cierren el fichero, el MGS borrará los datos. Aun así, si hubo algún tipo de error en los servicios o la red y el fichero no se cierra correctamente, el módulo del Cliente seguirá mandando señales de close al MDS.

2.5. CephFS

2.5.1. Arquitectura

CephFS es un sistema de archivos distribuido basado en Reliable Autonomic Distributed Object Store (RADOS) y parte del Ceph Storage Cluster, que es la base para todo el software Ceph. CephFS realiza las operaciones de I/O leyendo desde y escribiendo al cluster [9]. Sus servicios son:

- **Ceph Monitor** Se encarga de monitorizar a los Object Storage Daemon (OSD), actuando como coordinador de las operaciones. Antes de escribir o leer, el cliente debe conectarse a un monitor para obtener la última versión del mapa del cluster. Para evitar SPOF, un mismo Ceph Storage Cluster puede (y debería) tener varios monitores distribuidos en diferentes discos.

- **OSD** Es el servicio que se encarga de almacenar datos, y va ligado a un disco físico.

- **Metadata Server Daemon (MSD)** Uno o varios de estos servicios son los encargados de gestionar el espacio de nombres. Puede optarse por tener uno activo y varios en standby preparados para sustituir al activo, o tener varios activos que se dividan el espacio de nombres o, preferentemente con cantidades grandes de datos, una combinación de ambas. El espacio de nombres es plano, sin jerarquía.

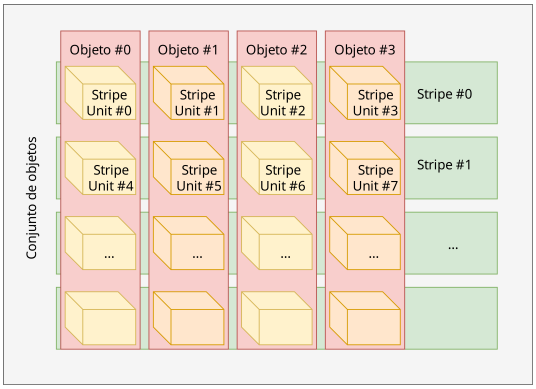
2.5.2. Distribución, particionado y replicación

El monitor implementa el algoritmo Controlled Replication Under Scalable Hashing (CRUSH). CRUSH es usado tanto por los Clientes como por los OSD, y calcula en que OSD debe almacenarse cada bloque de datos. Su comportamiento puede ser modificado por el administrador del sistema de acuerdo a sus necesidades, como categorizar y dar preferencia a los OSD (por ejemplo, catalogar discos Solid State Drive (SSD) como los más rápidos) o evitar que la replicación se haga en discos demasiado próximos. Lo más atractivo de CRUSH es que es completamente descentralizado: no necesita una tabla de búsqueda central, y los clientes interactuan directamente con los OSD para las operaciones de I/O.

Por lo tanto el Ceph Storage Cluster está doblemente descentralizado: un cliente que quiera hacer I/O primero contacta con un Monitor de un cluster de monitores que le facilita el Cluster Map, y después contacta directamente con los OSD de los discos físicos que le han sido asignados por CRUSH.

Los datos que almacena CephFS se convierten en objetos RADOS virtuales, que a su vez se trasladan a ficheros físicos que se alojan en el disco. Un cierto número de estos objetos conforman un Placement Group. Varios Placement Groups se integran en una Pool. El objetivo de los PG como conjunto intermedio es facilitar la gestión de metadatos y de localización de los propios objetos, ya que es más simple con estos grupos que individualmente. Esta relación objeto-placement group la calcula CephFS mediante un hash del ID del objeto. Las Pools están diseñadas para agrupar un número predeterminado de Placement Groups con el mismo factor de replicación. De esta manera, cada Placement Group de la Pool se almacenará en varios OSD. En resumen, el algoritmo CRUSH mapea cada objeto a un Placement Group y cada Placement Group a un OSD. El hecho de tener este indireccionamiento facilita rebalancear los datos cuando se escala el sistema agregando nuevos discos, ya que el cliente no sabe directamente donde está cada objeto, lo que crearía una acoplación muy grande entre éste y el OSD.

Figura 2.5: Disposición de los bloques de datos en CephFS



En cuanto al particionado, se hace en varios niveles. Un archivo es partido en varios trozos de datos, que se conocen como Stripe Units. Varias de estas unidades forman un Stripe, que no es sino una forma de mantener el orden de las Stripe Units. Pero no se almacenan contiguas, sino salteadas en Objetos, que a su vez forman Conjuntos de Objetos, como se ha dibujado en la figura 2.5.

2.5.3. Flujo de datos

- **Lectura** El cliente contacta con un monitor del cluster de monitores que le reporta el OSD en el que está alojado cada bloque. Se realiza la lectura de manera paralela desde los discos primarios preferentemente.

- **Escritura** De igual manera, el cliente recibe el OSD en el que tiene que escribir cada bloque de un monitor del cluster de monitores, calculado por CRUSH. Este OSD primario escribe sus datos a su Journal y después lo escribe en su disco físico (OST). Para cumplir el factor de replicación, de manera paralela otros OSS escriben en su Journal y en su OST.

  Es importante comentar el Journaling de CephFS. A diferencia de los sistemas de archivos hasta ahora explorados, aquí no se usa el Journal como un registro de transacciones: se usa para escribir todos los datos que el cliente escribe. Los desarrolladores lo programaron así buscando primar la estabilidad y escalabilidad frente a la velocidad. La consecuencia directa es que, con la configuración por defecto, se escribe el doble de datos en cada disco físico, ya que el Journal es un archivo más de dicho disco.

Una primera idea para solventar esto podría ser dedicar un disco físico únicamente para hacer el Journaling de todos los OSD. Si dicho disco es un Hard Disk Drive (HDD), sería un mal arreglo debido a que el cabezal estaría continuamente moviéndose, teniendo en cuenta que esperamos muchos clientes usando el sistema. Lo ideal y recomendado sería usar discos SSD (uno por OSD o uno general), más veloces en escritura y tiempo de acceso, para alojar la partición del archivo Journal y paralelizar su escritura. En nuestra configuración hemos puesto dicha partición en los primeros sectores del disco (los cilindros más afuera), ya que teóricamente son los más rápidos, pero teniendo en mente que es más un pequeño parche que una solución real a la caída de rendimiento de escritura.

- **Borrado** En CephFS los datos se borran de manera perezosa, ya que el cliente generaría mucho tráfico de red enviando un mensaje de borrado por cada bloque del archivo, más aun si éstos están replicados. Para evitar también una posible pérdida de rendimiento, los archivos se marcan como borrados en el MDS y se van liberando sus bloques según vaya siendo necesario.

2.6. Lustre

2.6.1. Arquitectura

Los servicios con los que cuenta Lustre [10] son similares a los de BeeGFS, en concreto:

- Uno o varios **MGS**, que guardan la configuración persistente del sistema en un Management Target (MGT). También se encargan de propagar información como reinicios.
- Los **MDS** manejan el espacio de nombres. La jerarquía y demás información de cada directorio se contienen en un MDT. Se encargan de controlar el alojamiento de nuevos bloques en los OSS y de la apertura y cerrado de archivos, y otras operaciones del espacio de nombres.
- Los **OSS** se encargan del almacenamiento de los datos como tal. Cada uno se empareja con uno o varios OST. En Lustre, generalmente dos OST se usan de manera sincronizada para mantener la alta disponibilidad, con un Redundant Array of Independent Disks (RAID), a nivel físico. Actualmente Lustre no dispone de replicación a nivel de su propio software estable, pero esta característica ya está en desarrollo por lo que sería esperable que aparezca en próximas versiones.

Cabe nombrar también Lustre Networking (LNet), un protocolo de comunicación propio de Lustre diseñado para ser ligero y usado por todos los componentes del sistema.

2.6.2. Distribución, particionado y replicación

El particionado se puede establecer a nivel de archivo, carpeta o sistema de archivos. Las tres variables que se han de manejar son el Stripe Count (número de discos donde hacer la distribución), el Stripe Offset (disco en el que empezar la secuencia Round-Robin de escritura) y el Stripe Size (tamaño del bloque de datos).

Para el Stripe Count podemos usar el valor -1, que se traduce en usar todos los discos disponibles. El Stripe Offset usa por defecto el valor 1, lo cual puede ser un problema ya que causaría que los datos siempre se empiecen a escribir en el mismo disco, formando un cuello de botella. Por tanto parece recomendable usar también el valor -1, dándole a Lustre la capacidad de que elija el que más le convenga. Como se comentó en la sección anterior, aún no dispone de replicación a nivel software.

2.6.3. Flujo de datos

- **Lectura** El cliente pide al MDS la lista de OST que tienen los bloques del archivo que quiere leer. Se comunica directamente con cada OSS hasta que haya obtenido todos los datos, y entonces envía una confirmación al MDS para que haga accesible el archivo a otros clientes.

- **Escritura** Cuando un cliente quiere escribir, contacta con un MDS para que le reporte la lista de OST donde depositar los bloques de datos. Entonces y de manera paralela, el cliente contacta con cada OSS y efectúa la escritura, sin depender más del MDS.
- **Borrado** Los objetos relacionados al archivo se borran del almacenamiento de metadatos y sus bloques se marcan como libres, considerándose así el archivo borrado.

2.7. Conclusión

Hemos visto que, aunque parecidos por encima, los cinco sistemas de archivos tiene sus diferencias de diseño y funcionamiento. De cara a las pruebas del próximo capítulo, podríamos intuir ciertos comportamientos que deberemos ver si se cumplen. En general, sería idóneo un rendimiento siempre estable frente a tamaño de archivo, y en menor medida frente al tamaño del bloque que elijamos (suponiendo que cada archivo tenga uno diferente).

Finalmente, una aclaración, en cuanto a la instalación y funcionamiento de estos sistemas. Tanto HDFS como CephFS destacan excepcionalmente por su facilidad de instalación y unas guías y wikis muy completas. En este apartado Lustre es de lejos el peor, con poca información y pobremente presentada. En cuanto a la administración del sistema, todos tienen su conjunto de comandos más o menos equiparables, sin ser ninguno especialmente complicado.

3 Pruebas de rendimiento

3.1. Introducción

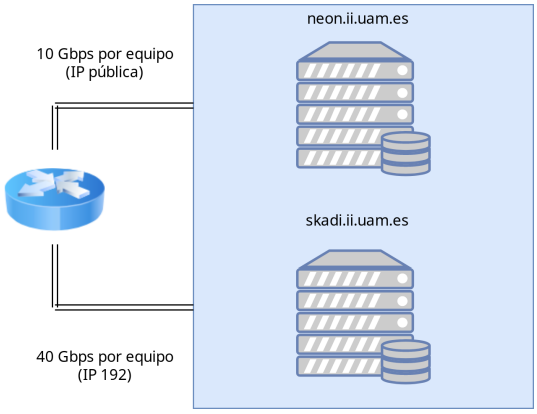
En el capítulo anterior hemos descrito los cinco sistemas de archivos distribuidos, así como sus ventajas y carencias. Ahora efectuaremos diferentes pruebas de I/O, para comprobar si la descripción teórica se ajusta a la realidad, así como analizar el rendimiento de acuerdo a varios parámetros.

El software que usaremos para ello es flexible input-output (fio) de Jens Axboe[I], una completa herramienta que permite generar cargas de trabajo para probar el rendimiento de un sistema de archivos, teniendo en cuenta una multitud de parámetros. fio se aprovecha de los sistemas de ficheros que cumplen el estándar Portable Operating System Interface (POSIX) para realizar las pruebas de rendimiento a través del acceso I/O nativo proporcionado por el sistema. Por sus características de sistema virtual, HDFS no es POSIX-compatible y es necesario usar `libhdfs`, que conecta fio con el sistema de archivos. Los otros cuatro, al estar montados como una carpeta POSIX más, no necesitan nada especial para ser probados, aunque para GlusterFS también se ha usado una librería `libgfapi`[II], que permite interacción entre fio y los servicios de dicho sistema de archivos sin necesidad de usar un punto de montaje.

3.1.1. Topología del cluster

El cluster sobre el que operaremos consta de dos máquinas con una distribución CentOS 7.4, con procesadores Intel Xeon E5-2620 v3 a 2.40GHz y 2 nodos NUMA de 32GB de RAM. Cada una cuenta con 7 discos HDD operativos, cuya velocidad de rotación es de 7200rpm y 3 terabytes de capacidad de almacenamiento. En la figura 3.1 se presenta un diagrama de red de este cluster, con los nombres de las máquinas.

Figura 3.1: Diagrama de red del cluster



La máxima que seguiremos es instalar los servicios de gestión y el cliente en la máquina `neon.ii.uam.es`, y los servicios de metadatos y almacenamiento en ambos nodos.

3.1.2. Definición de las pruebas

Hemos definido cuatro pruebas con el objetivo de abordar cuatro casos de uso, son las siguientes:

1. **Prueba primera (tamaño de las peticiones de datos)** Con la primera prueba queremos comprobar la respuesta del sistema de archivos con un tamaño de petición de datos cambiante. El caso de uso relacionado sería un usuario en un escenario tipo streaming o de procesamiento en lotes, donde pide una gran cantidad de datos de manera continua, y otro escenario contrario, tipo aplicación interactiva, donde el usuario hace peticiones muy pequeñas de datos, requiriendo por ello una latencia baja. En nuestra casuística de

[I] `http://fio.readthedocs.io/en/latest/`
[II] `http://staged-gluster-docs.readthedocs.io/en/release3.7.0beta1/Features/libgfapi/`

procesamiento de datos de red, estaríamos más bien ante el primer escenario, por lo que desearíamos mejor rendimiento ante peticiones grandes de datos.

2. **Prueba segunda (tamaño de bloque)** En esta segunda prueba, suponiendo un tamaño de petición de datos adecuado para nuestro caso, como se comentó arriba, el parámetro a variar será el tamaño de bloque de los sistemas de archivos (el tamaño de chunk, en terminología HDFS). Hay que notar que esta prueba no se ejecutará con GlusterFS ya que, como se comentó en la sección 2.3, por defecto el tamaño de bloque no lo elige el administrador del sistema sino que es dinámico. Buscamos comprobar sobre que rango de tamaños de bloque se comporta mejor el sistema, para obtener así un valor ideal de tamaño de bloque.

3. **Prueba tercera (tamaño de archivo)** Para esta prueba, modificaremos el tamaño de archivo, sabiendo el tamaño de petición de datos y de bloque adecuados a cada sistema de archivos. La finalidad de esta prueba es obtener una idea del rendimiento del sistema ante casos de uso con archivos tanto pequeños como grandes. Para nuestro problema, buscamos un rendimiento alto en archivos medianos, sobre los 512MB-4GB.

4. **Prueba cuarta (tamaño de las peticiones de datos, caso desfavorable)** Esta última prueba queremos enfocarla a un escenario donde el tamaño de petición de datos esté fijo a un tamaño muy bajo, y variar entonces el tamaño de archivo. La finalidad es comprobar sobre que valores se comporta mejor la aplicación en un caso del tipo interactivo, o de escritura y lectura en tiempo real.

Cuadro 3.1: Relación de valores usados en las pruebas con fio

Prueba	**Tamaño de petición de datos**	**Tamaño de bloque**	**Tamaño de archivo**
Prueba 1	*256KiB - 40MiB*	Por defecto del DFS	4GiB
Prueba 2	Mejor valor Prueba 1	*4KiB - 2GiB*	4GiB
Prueba 3	Mejor valor Prueba 1	Mejor valor Prueba 2	*128MiB - 64GiB*
Prueba 4	400B	Mejor valor Prueba 2	*128MiB - 64GiB*

Las configuraciones usadas para las pruebas se describen en la tabla 3.2, Para cada una de ellas se harán al menos cuatro muestras, a fin de evitar valores irreales, y se usará la media de estas cuatro. Como también se comenta, se han realizado pruebas sin usar ningún sistema distribuido intermedio, etiquetadas como `Raw` en las gráficas. Estas pruebas se han realizado ejecutando fio directamente sobre un disco cualquiera, y se ha multiplicado por el número de discos para obtener una guía del rendimiento máximo teórico.

Cuadro 3.2: Correspondencias de la leyenda de las gráficas de resultados

Nombre	**Descripción**
`raw`	Prueba ejecutadas directamente sobre FS del disco, sin DFS intermedio.
`replication1, replication2`	Configuración por defecto y replicación 1 o 2.
`optim_replication1, optim_replication2`	Ajustes del apéndice A y replicación 1 o 2.
`distr_strip_fuse, distr_strip_repl_fuse`	Configuración por defecto y replicación 1 o 2 (GlusterFS con FUSE, ver figuras 2.3 y 2.4).
`distr_strip_gfapi, distr_strip_repl_gfapi`	Configuración por defecto y replicación 1 o 2 (GlusterFS con librería `libgfapi`, ver figuras 2.3 y 2.4).

3.2. Gráficas de resultados

En esta sección se presentan los resultados así como una pequeña explicación de cada figura.

3.2.1. Prueba primera

Resultados de la prueba (escritura)

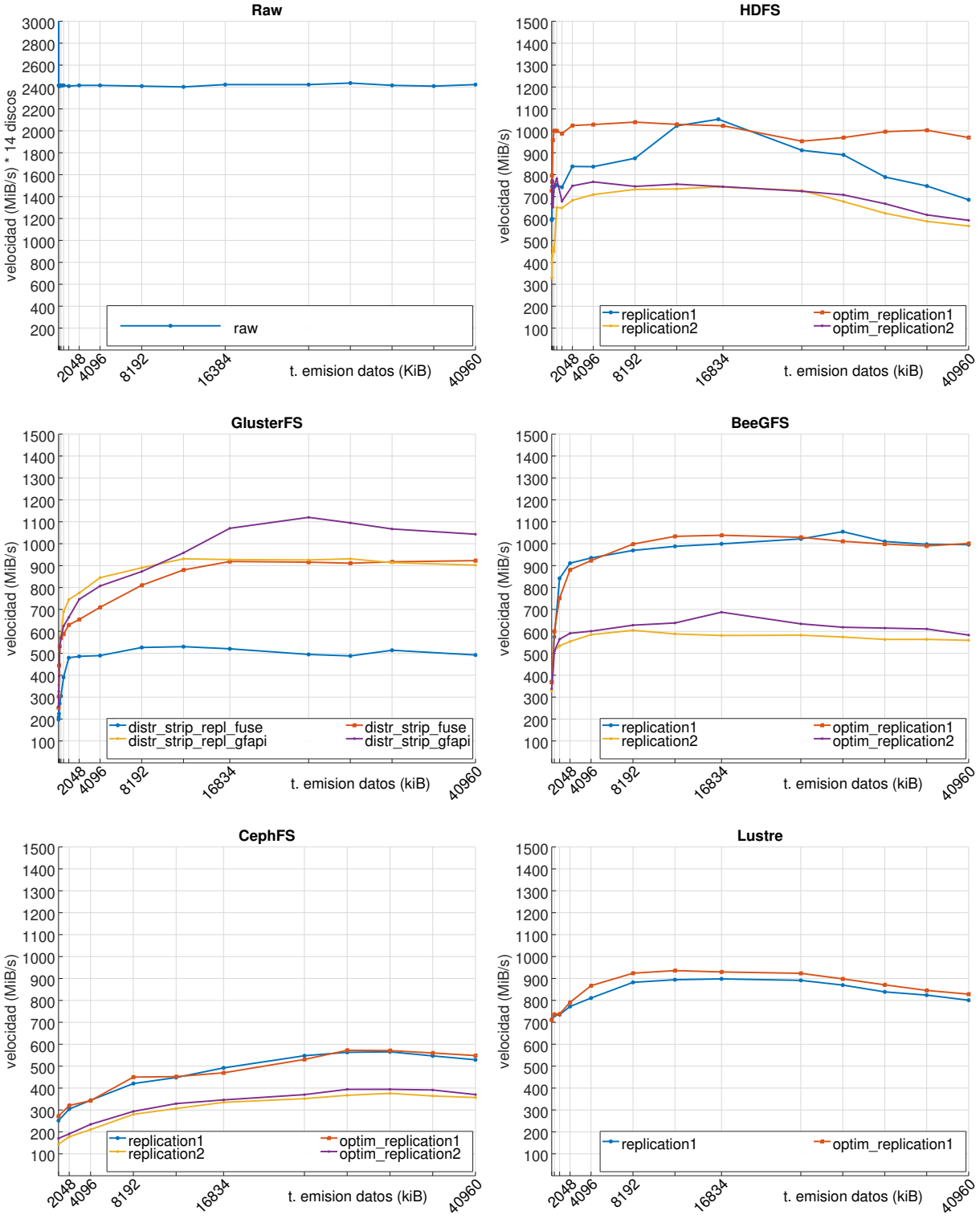


Figura 3.2: Gráficas de resultados de la prueba 1 (write)

Vemos un comportamiento esperado: en tamaños de petición de datos muy pequeños, el rendimiento es malo, pero a partir de un umbral, se mantiene bastante estable. En los cinco sistemas distribuidos se encuentra entre los 2048 y 4096 KiB, aunque en BeeGFS se muestra mucho más estable en todo momento, mientras que en CephFS, por su característico Journaling[11], las gráficas no muestran claramente el mismo comportamiento logarítmico ni rendimiento.

Resultados de la prueba (escritura aleatoria)

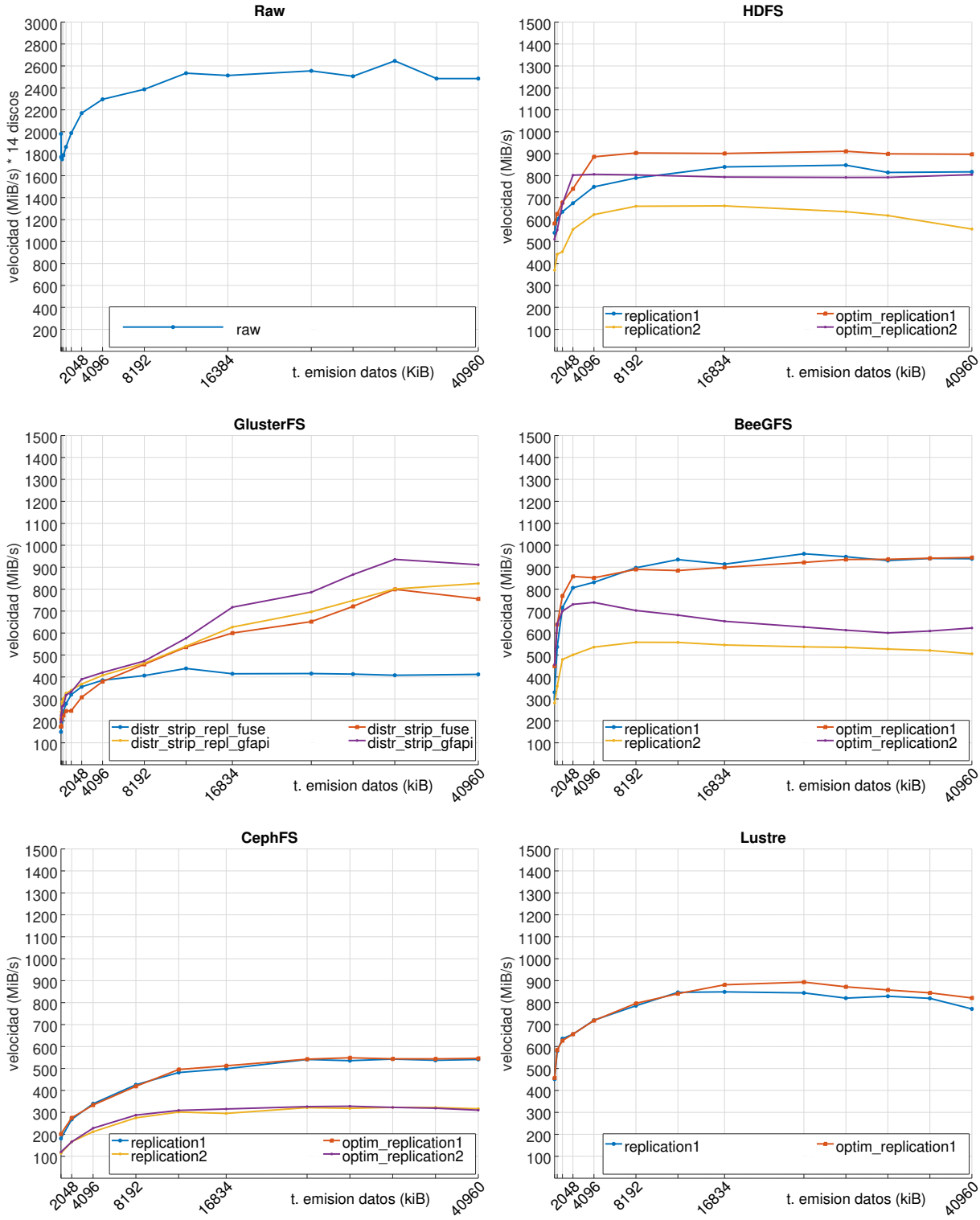


Figura 3.3: Gráficas de resultados de la prueba 1 (randwrite)

Las gráficas son muy parecidas a sus homólogas secuenciales, pero con menos rendimiento al sufrir más el software con el acceso aleatorio, debido a la necesidad de mover más constantemente el cabezal. Este estudio[12] obtiene que un HDD es un 99.27 % más lento que un SSD en accesos aleatorios. Notamos varios puntos interesantes: Lustre y GlusterFS mantienen los resultados excepto en los tamaños de bloque pequeños, más claramente este último. Por otro lado, BeeGFS pierde poco rendimiento, al igual que CephFS, mientras que HDFS consigue mantener el rendimiento con replicación 2 y no perder más de un 10 % con replicación 1.

Resultados de la prueba (lectura)

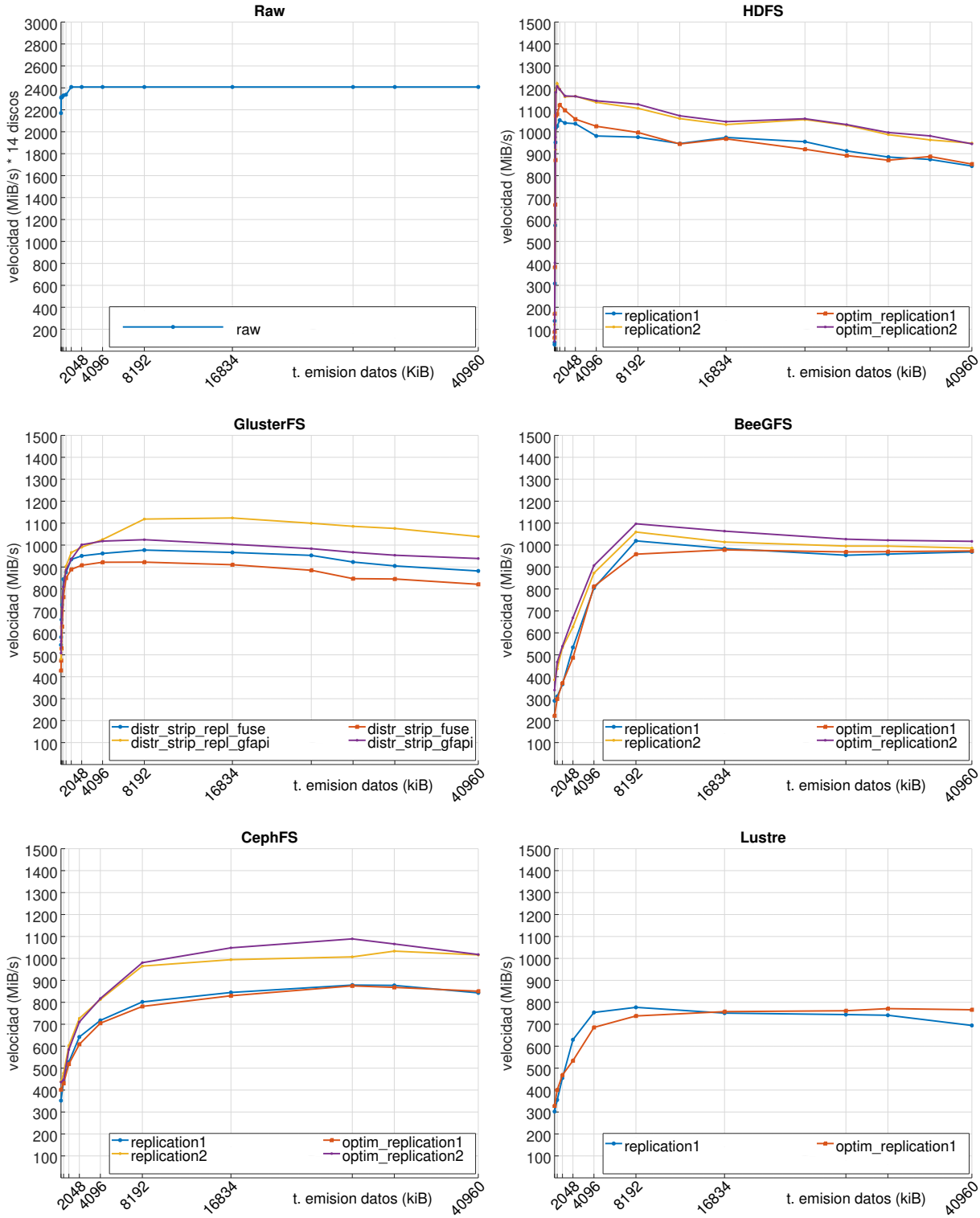


Figura 3.4: Gráficas de resultados de la prueba 1 (read)

En la lectura secuencial, podemos apreciar unos comportamientos logarítmicos similares, rondando igualmente los 2048-4096KiB como punto de estabilización. Cabe mencionar el caso de HDFS, donde se observa que bloques cada vez más grandes hacen que disminuya ligeramente su rendimiento (aunque en GlusterFS y CephFS también ocurre, de manera más sutil). También es interesante que en Lustre y BeeGFS el rendimiento es muy similar en todos los casos, siendo CephFS y, algo menos, GlusterFS los únicos donde la falta de replicación lastra claramente el rendimiento en lectura.

Resultados de la prueba (lectura aleatoria)

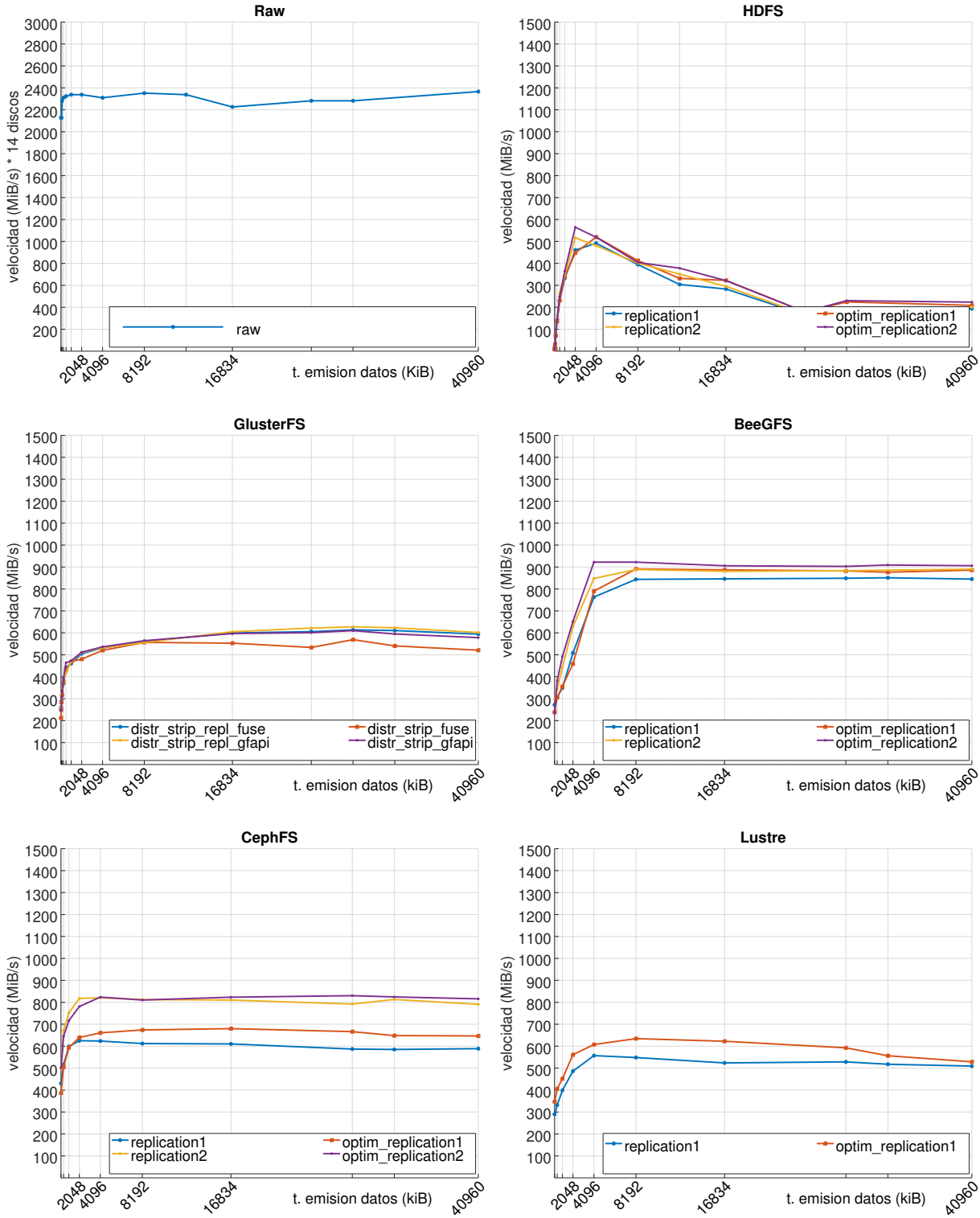


Figura 3.5: Gráficas de resultados de la prueba 1 (randread)

Esta prueba es especialmente dura para HDFS que, una vez alcanzado el punto donde los otros sistemas se estabilizan, sufre una caída de rendimiento. GlusterFS gestiona mal la aleatoriedad, ya que incluso con un factor de replicación 2, sus resultados son inferiores a CephFS y BeeGFS con factor de replicación 1. Éste último y Lustre parecen conseguir mantenerse sin demasiada pérdida de rendimiento.

3.2.2. Prueba segunda

Cabe recordar que en esta prueba no entra GlusterFS, como se explicó en la sección 3.1.2.

Resultados de la prueba (escritura)

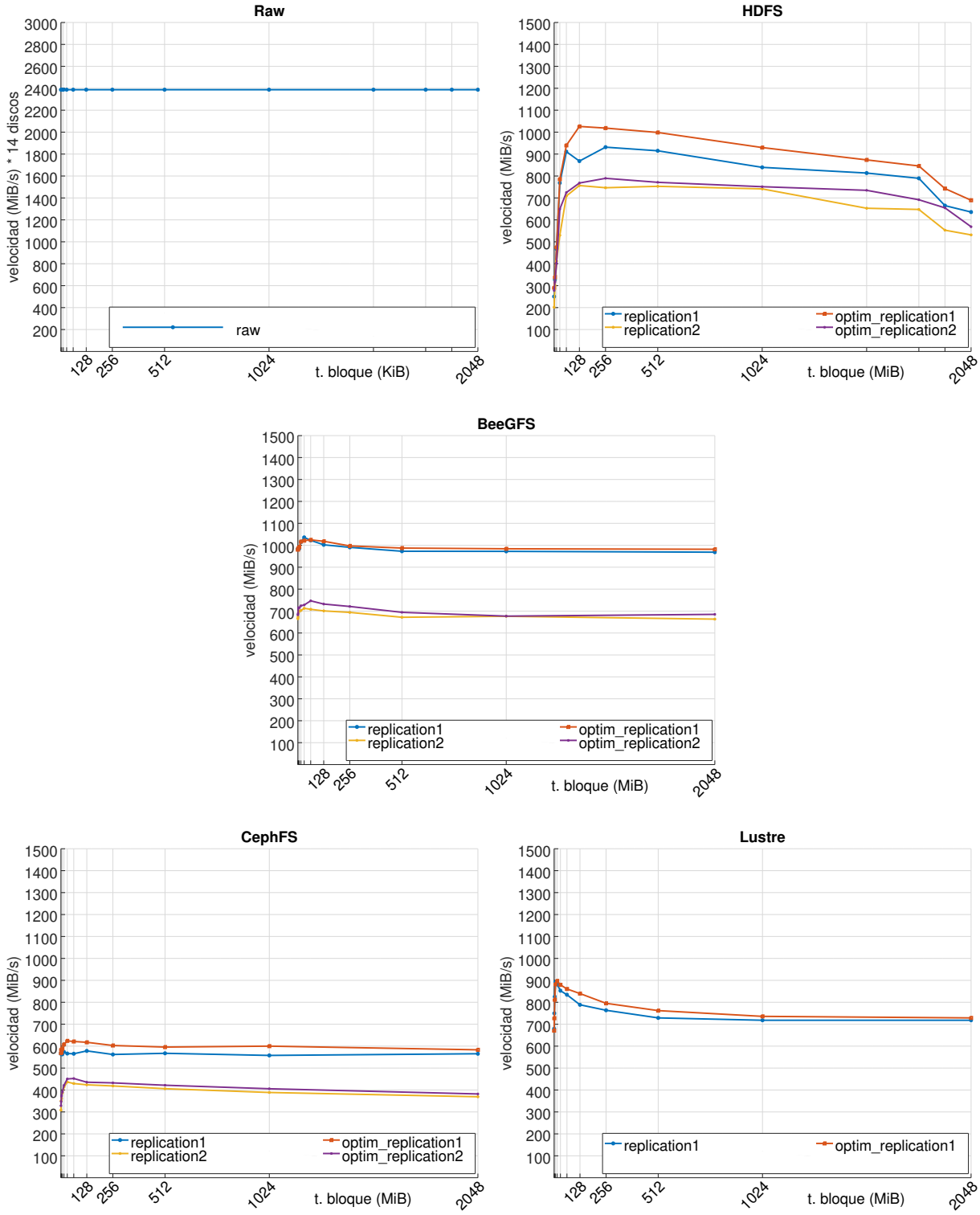


Figura 3.6: Gráficas de resultados de la prueba 2 (write)

El tamaño de bloque, a partir de 32MiB en todos los casos menos en HDFS, mantiene un rendimiento estable. En HDFS, alcanza dicho punto en 64MiB (la configuración por defecto), y sufre una caída en tamaños más grandes, acentuándose especialmente a partir de 1792MiB. Para tamaños de bloques muy pequeños, el rendimiento es algo peor debido un mayor overhead para un bloque pequeño que uno grande[13].

Resultados de la prueba (escritura aleatoria)

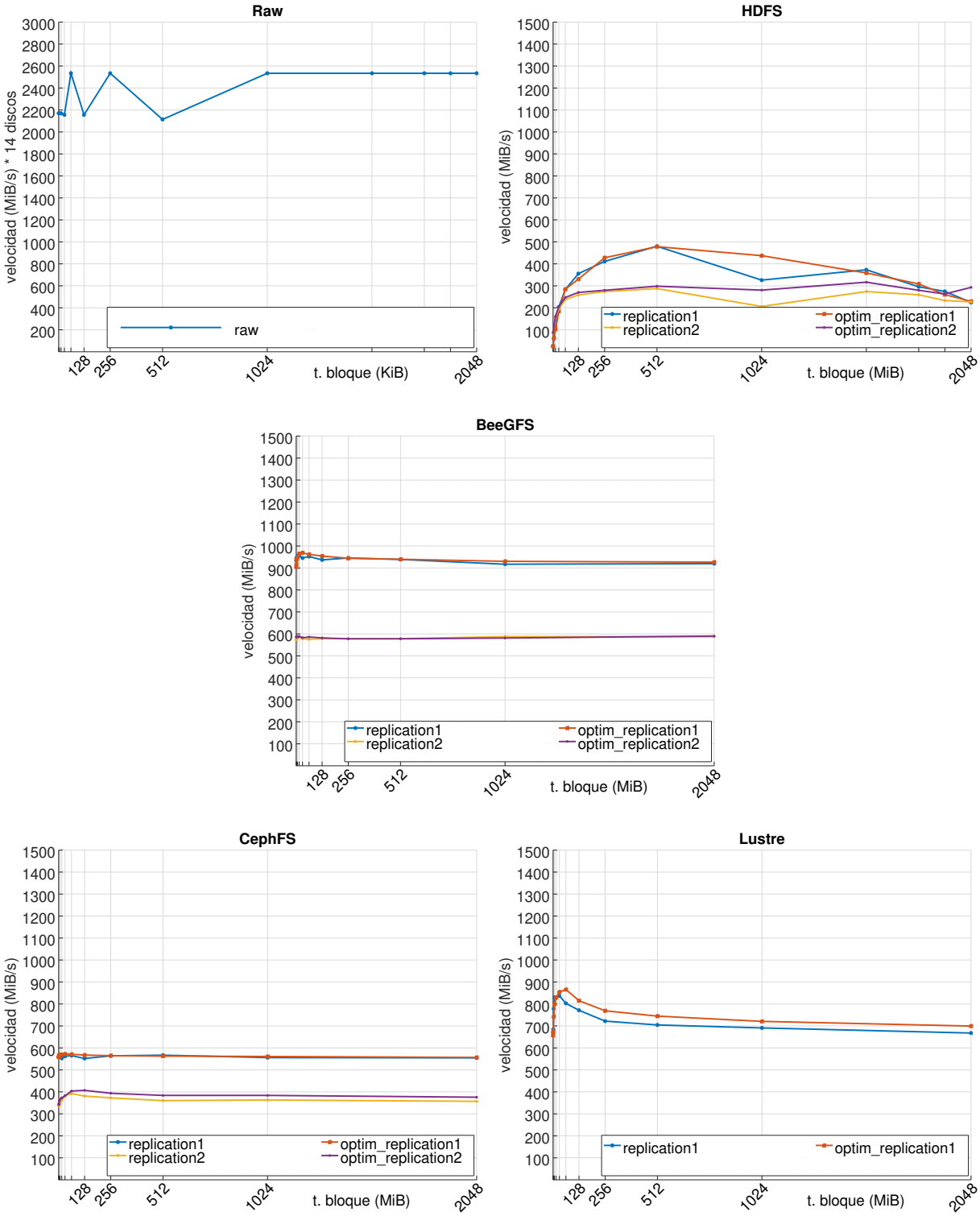


Figura 3.7: Gráficas de resultados de la prueba 2 (randwrite)

Nuevamente, HDFS sufre especialmente con accesos aleatorios. Los demás mantienen un comportamiento parecido, pero un rendimiento tope algo menor, más acentuado en BeeGFS con replicación. Tanto en este caso como en el anterior, ahora se puede observar mucho más claramente una diferencia de rendimiento entre ambos factores de replicación, más claramente en CephFS y BeeGFS.

Resultados de la prueba (lectura)

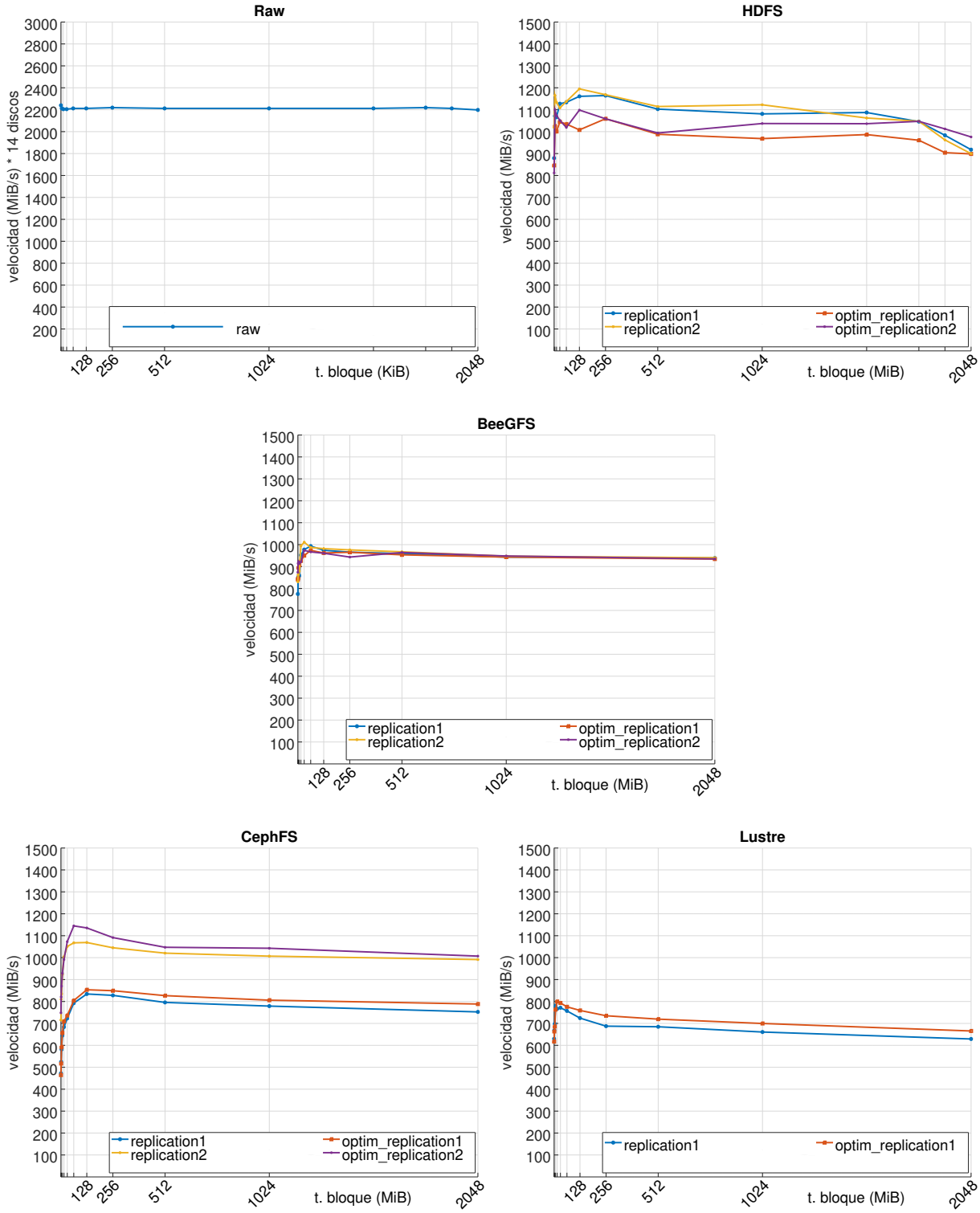


Figura 3.8: Gráficas de resultados de la prueba 2 (read)

Para la lectura, vemos comportamientos similares. Es interesante que el rendimiento de BeeGFS, a diferencia de en las escrituras, se ha igualado en los cuatro casos, lo que parece significar que gestiona mejor la lectura. HDFS obtiene unos resultados similares, mientras que CephFS obtiene resultados parecidos con factor de replicación 2, pero algo peores cuando éste vale 1, siendo parecidos a los valores de Lustre.

Resultados de la prueba (lectura aleatoria)

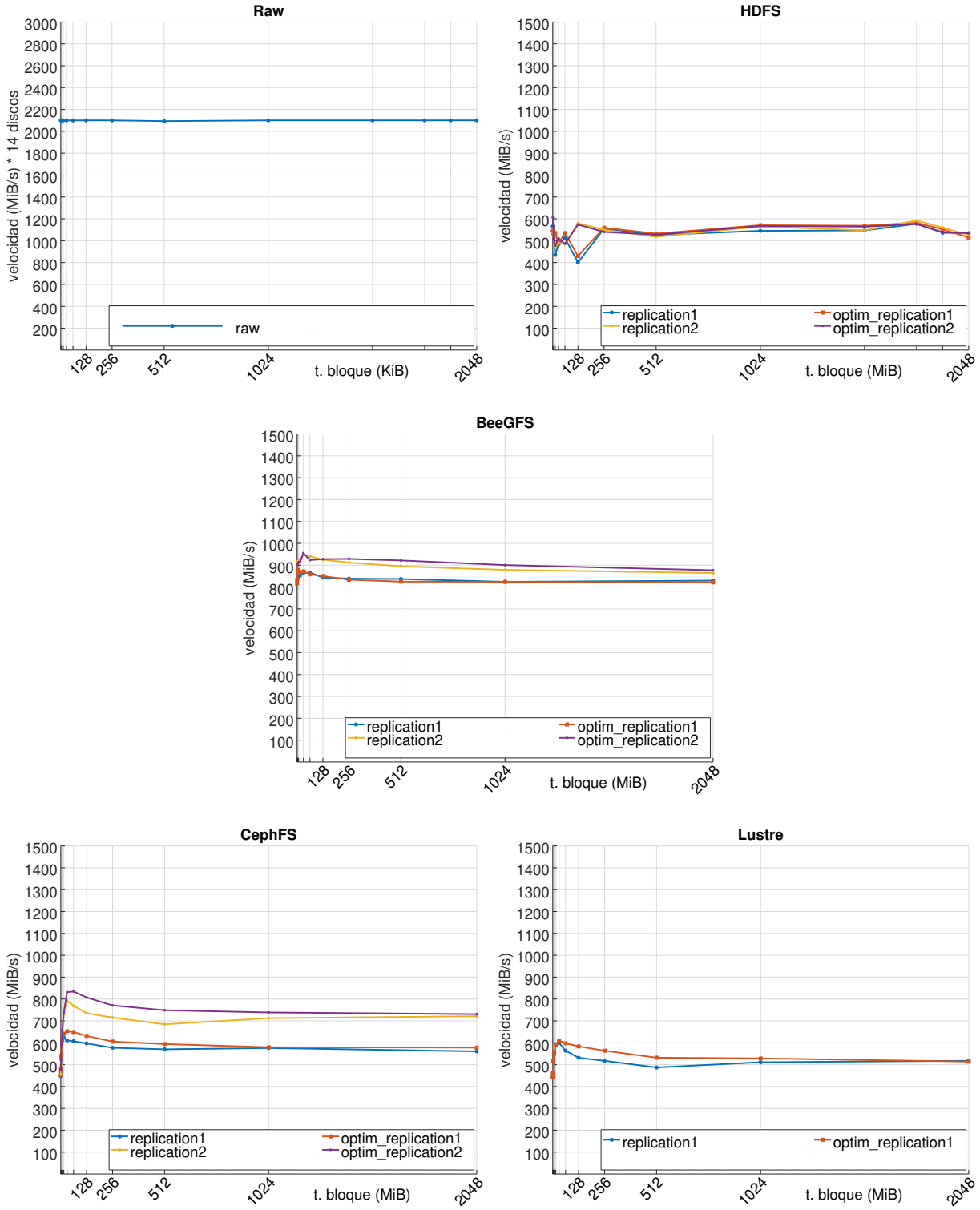


Figura 3.9: Gráficas de resultados de la prueba 2 (randread)

Observamos nuevamente, aunque menos acentuado, el fenómeno anterior en BeeGFS. La gráfica de CephFS mantiene la forma, aunque pierde una parte de rendimiento, igual que Lustre. Vemos otra vez una gran pérdida de rendimiento en HDFS con I/O aleatorio, fenómeno mucho más acentuado que en cualquiera de los demás sistemas de archivos. Un motivo es la cantidad de datos transferidos y descartados debido a su arquitectura esencialmente de acceso secuencial, así como la creación de una conexión Transmission Control Protocol (TCP) en cada acceso, aunque se trate de muchos accesos al mismo nodo[14, figuras 1 y 2].

3.2.3. Prueba tercera

Resultados de la prueba (escritura)

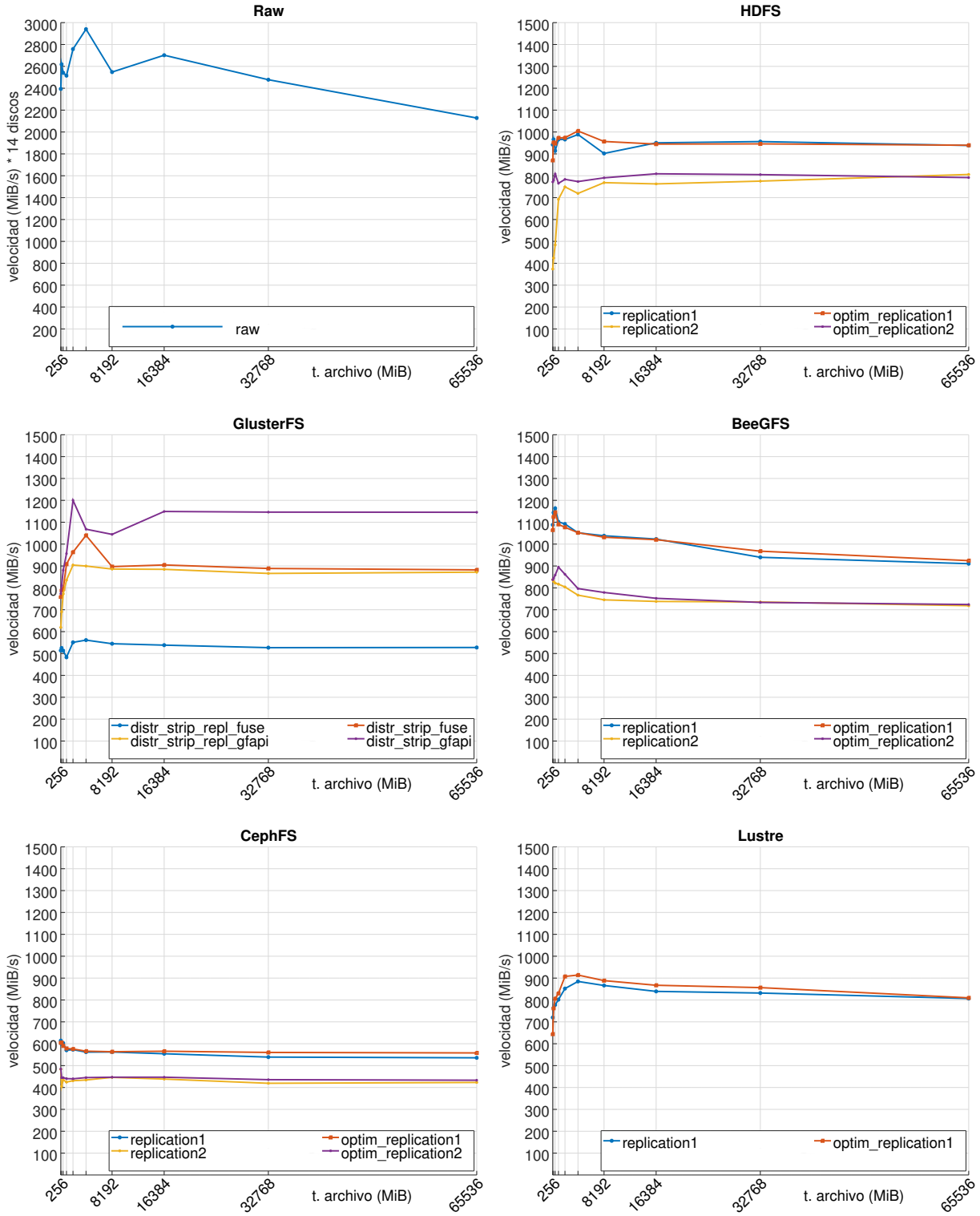


Figura 3.10: Gráficas de resultados de la prueba 3 (write)

HDFS es el único que empieza con un rendimiento, bastante bajo, de 400MiB/s en archivos pequeños, pero a partir los 4GiB se estabiliza. Este se debe a la característica peculiar de que el NameNode mantiene toda la lista de archivos y carpetas en memoria, ocupando cada entrada 150 bytes[15], lo cual afecta al rendimiento con un gran número de pequeños archivos. De igual manera se observa este fenómeno en Lustre, aunque empieza en un valor algo más alto sobre los 600MiB/s. Al contrario, GlusterFS, CephFS y BeeGFS mantienen un comportamiento algo más plano frente a todos los tamaños de archivo, viéndose en el último incluso un mejor rendimiento en los primeros valores.

Resultados de la prueba (escritura aleatoria)

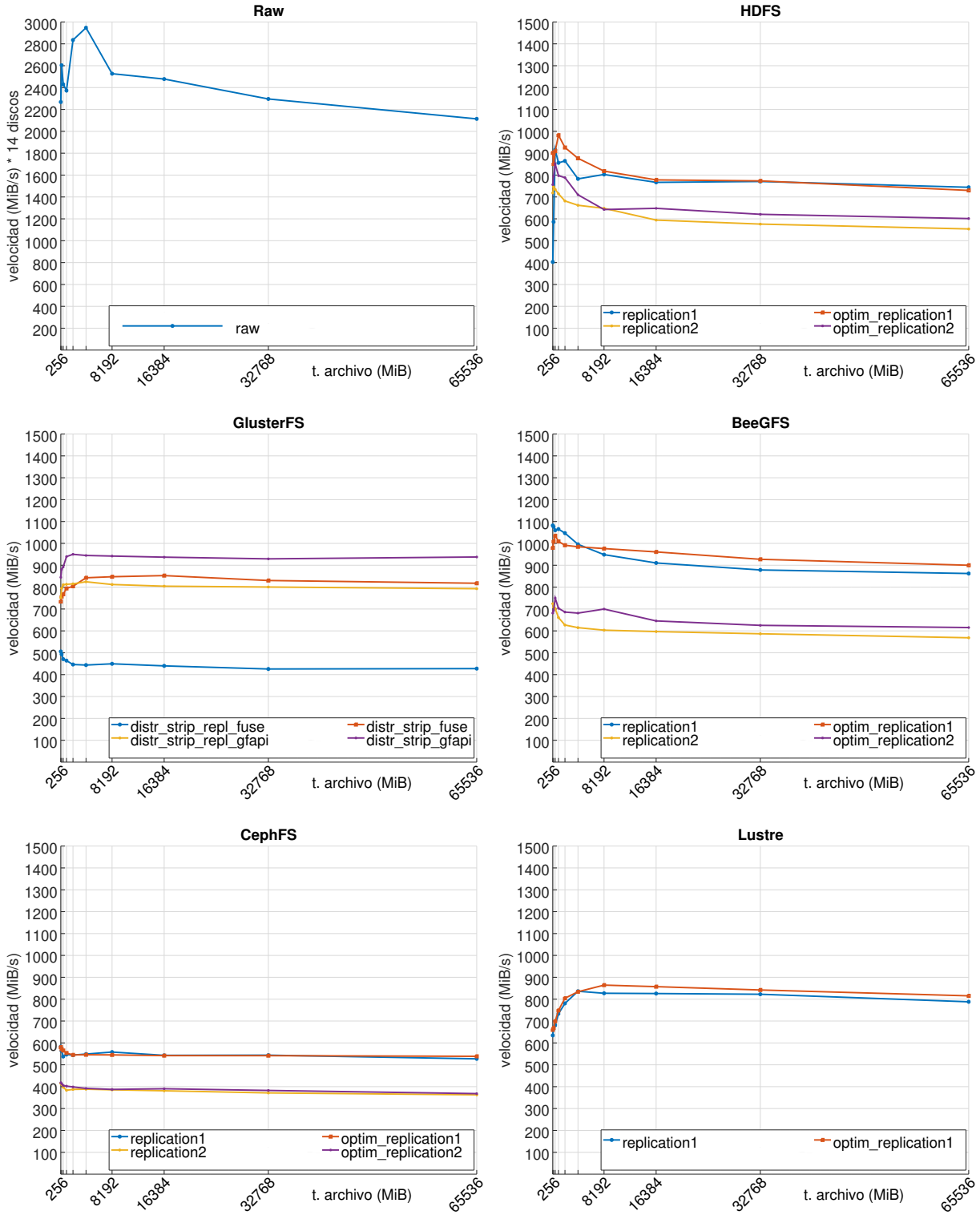


Figura 3.11: Gráficas de resultados de la prueba 3 (randwrite)

Esta es la única prueba de acceso aleatorio donde HDFS mantiene un rendimiento similar a sus competidores, que siguen con velocidades similares, algo más bajas en Lustre, GlusterFS y especialmente en BeeGFS con replicación 2 (algo que ya vimos en la prueba 2 de escritura aleatoria). CephFS mantiene un comportamiento muy similar a la escritura secuencial.

Resultados de la prueba (lectura)

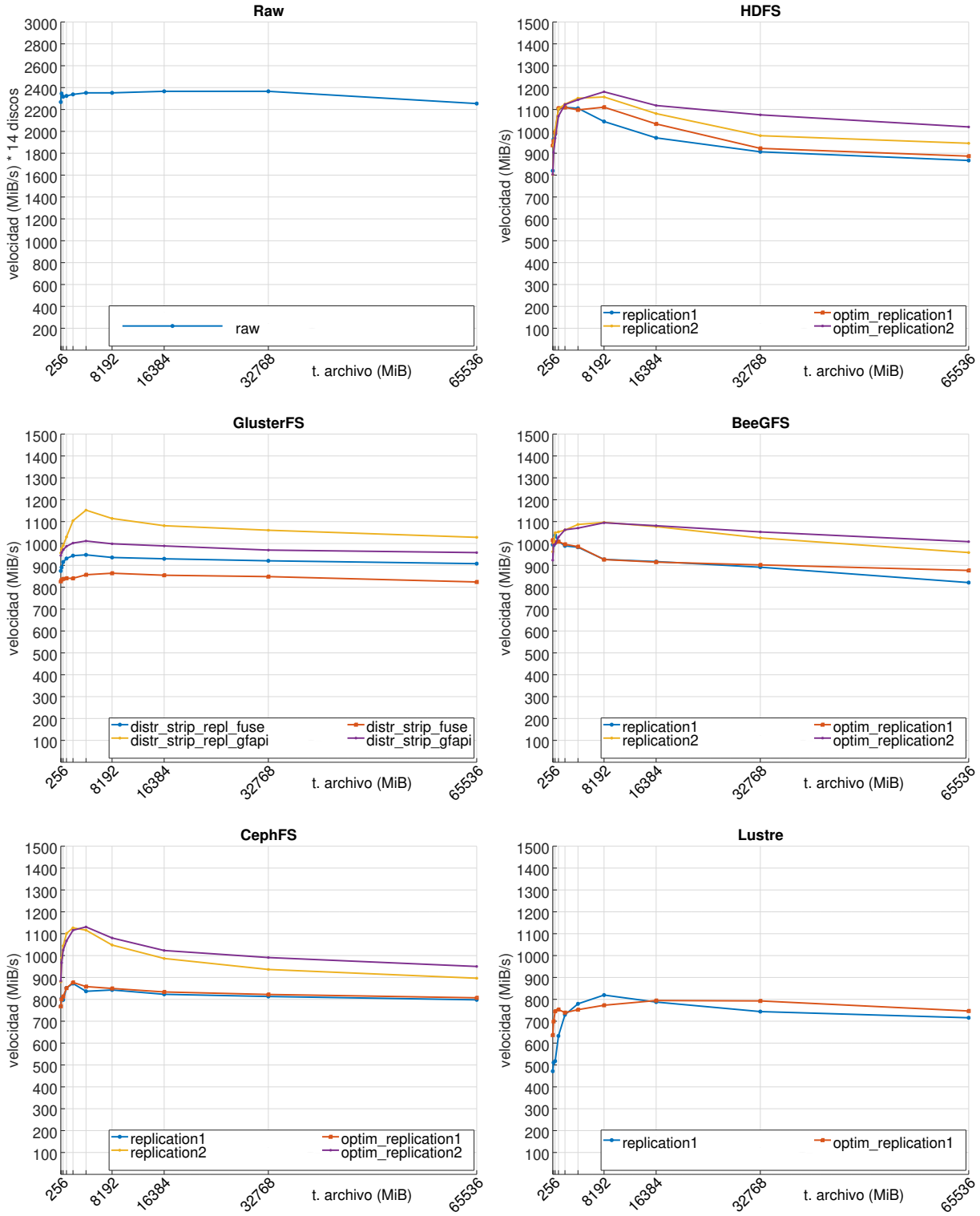


Figura 3.12: Gráficas de resultados de la prueba 3 (read)

En la lectura sucede lo mismo con HDFS, velocidad menor inicial que se empieza a estabilizar sobre los 4GiB, algo que vuelve a suceder en Lustre, pero más acentuado (la subida hasta la parte plana de la gráfica es más larga). BeeGFS mantiene un rendimiento parecido al de la escritura, mientras que en GlusterFS los cuatro casos tienen resultados mucho más homogéneos que en la escritura, manteniendo un rendimiento tope igualmente sobre los 1100MiB/s.

Resultados de la prueba (lectura aleatoria)

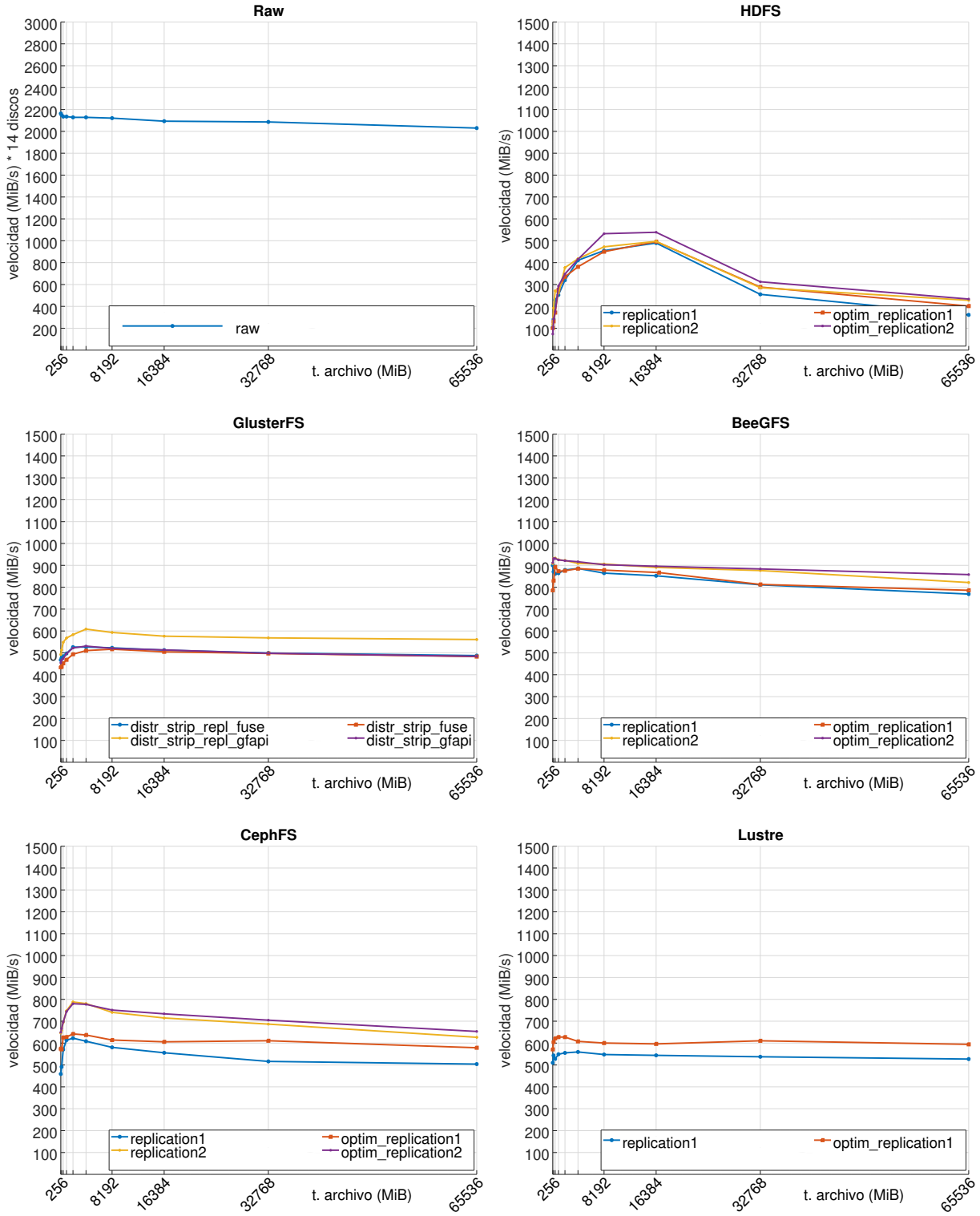


Figura 3.13: Gráficas de resultados de la prueba 3 (randread)

Nuevamente observamos que CephFS pierde más rendimiento al leer de manera aleatoria que al escribir, lo que también podemos ver ahora en GlusterFS. Otra vez HDFS tiene un rendimiento comparativamente peor que sus rivales. Lustre pierde poco, como ya vimos en la escritura aleatoria, y BeeGFS pierde más con replicación 2 que con replicación 1, dado que en este último caso los valores se mantienen casi al mismo nivel.

3.2.4. Prueba cuarta

Resultados de la prueba (escritura)

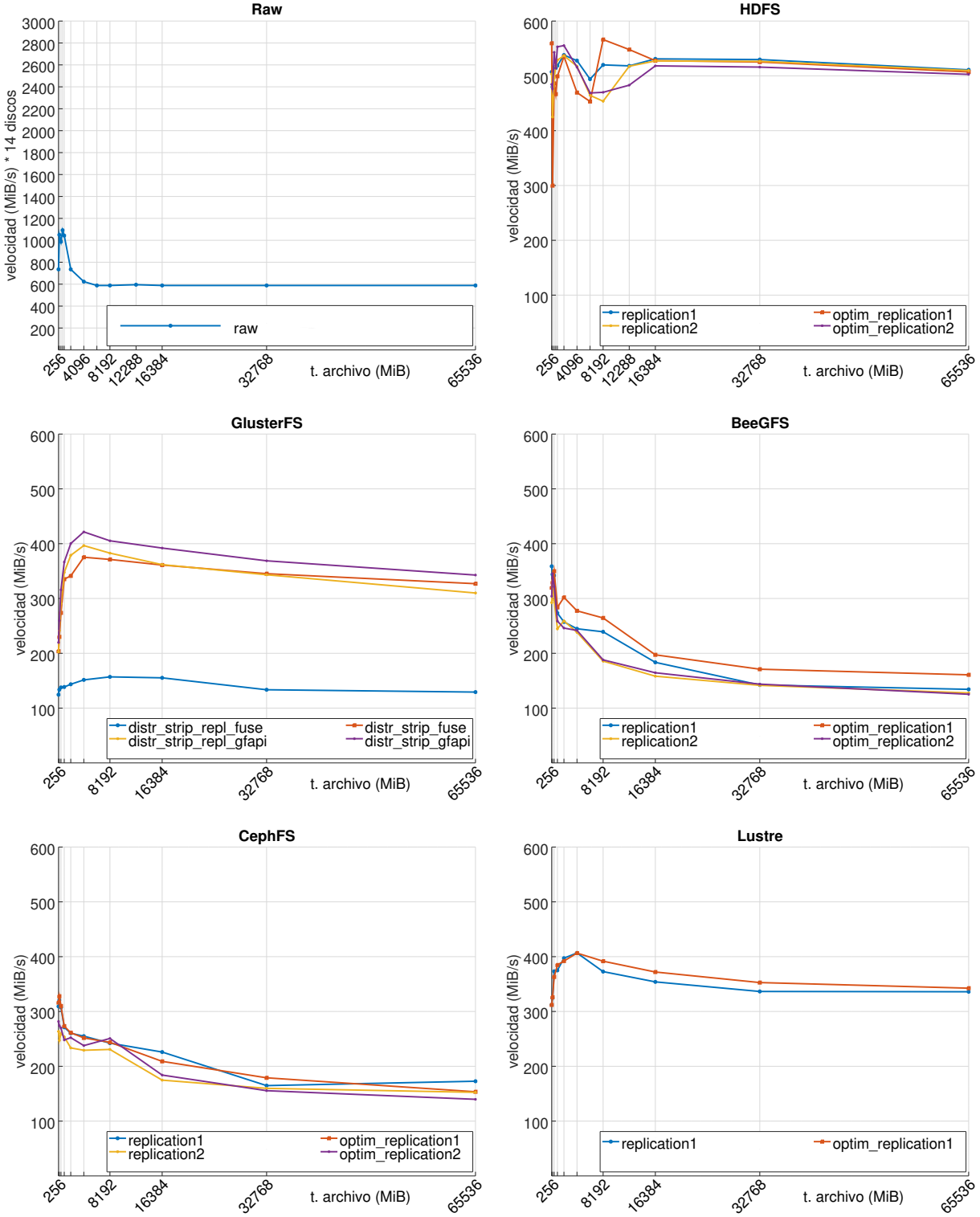


Figura 3.14: Gráficas de resultados de la prueba 4 (write)

En todos los casos, el rendimiento es mucho peor que con el tamaño de emisión de datos elegido según la prueba primera, aunque en HDFS el rendimiento se acerca mucho al tope que vemos en Raw. Le sigue de cerca Lustre, mientras que los demás mantienen velocidades decrecientes con mayores tamaños de archivo. El peor es claramente GlusterFS con replicación y acceso por FUSE.

Resultados de la prueba (lectura)

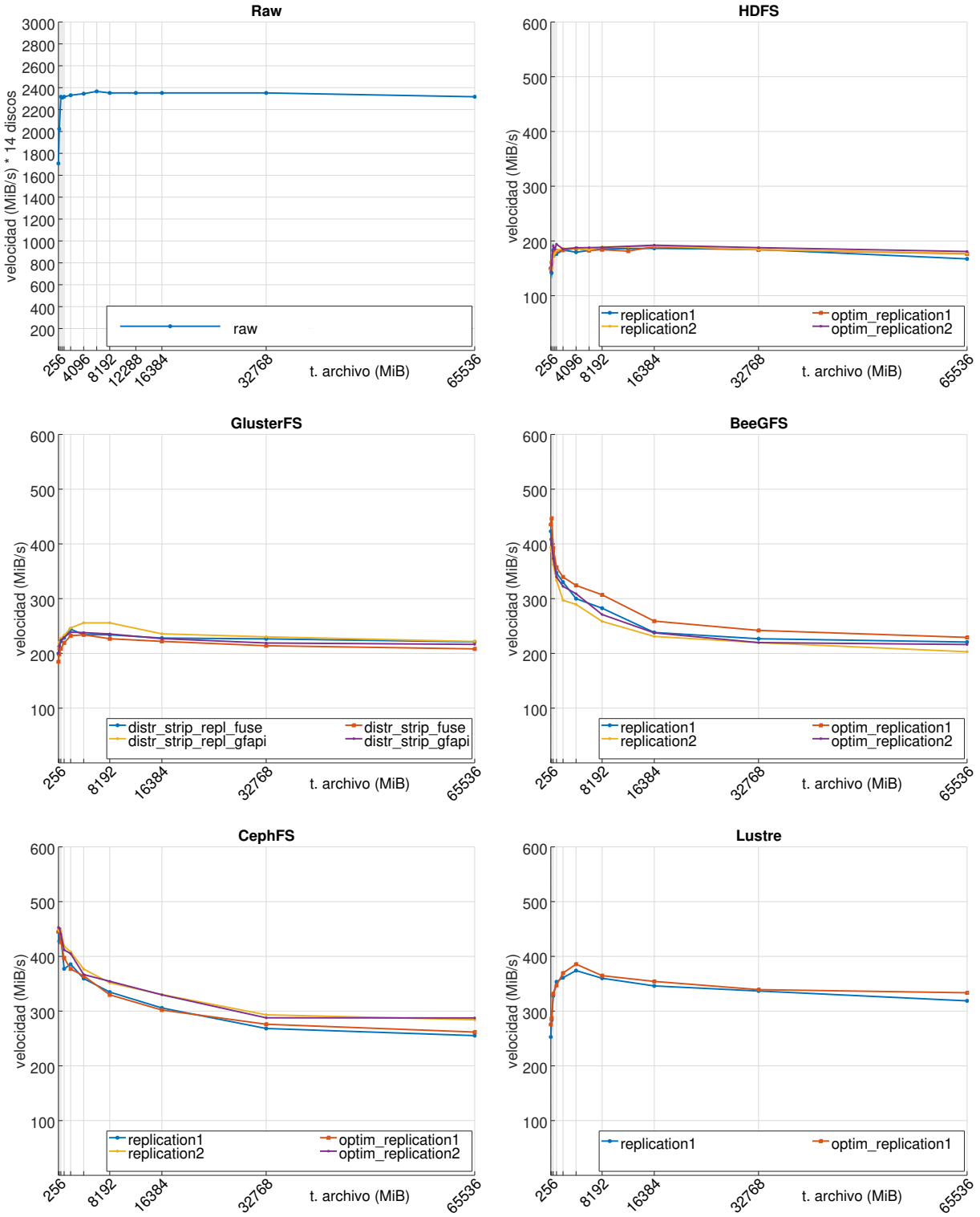


Figura 3.15: Gráficas de resultados de la prueba 4 (read)

En este caso HDFS no consigue rendimientos tan aceptables, GlusterFS también ve reducida a la mitad su velocidad. Los demás, en cambio, mantienen valores similares a los de escritura, en CephFS incluso algo mejores.

3.3. Análisis de resultados

Como hemos podido ver a lo largo de estas pruebas, el sistema de archivos distribuido *perfecto* no existe, pero si es posible ver como cada uno de ellos destaca en ciertos aspectos. Un fenómeno general que hemos observado son los factores de replicación favorables. La escritura tiene mejor rendimiento con factor de replicación 1 y la lectura, en cambio, con factor de replicación 2. La explicación lógica es que la escritura replicada necesita hacer el doble de I/O por lo que, aunque esto se intente paralelizar lo más posible, la velocidad decae. En cambio, la lectura se beneficia de tener bloques replicados ya que, si no puede acceder a un disco al estar éste en uso, puede ir a otro donde esté dicho bloque replicado.

En la prueba primera, el comportamiento es más o menos esperable: obtenemos mejor rendimiento con peticiones de mayor cantidad de datos, en vez de enviar pequeñas cantidades del archivo en varias veces, teniendo en cuenta que esto implica más pequeños accesos a metadatos y a los diferentes discos donde alojar los datos. Esto también implicaría más mensajes de red para alojar la misma cantidad de datos y una posible saturación de ésta. En la lectura sucedería algo parecido, se tendría que contactar múltiples veces con los diferentes servicios para obtener los mismos datos. En resumen, todos estos DFS parecen más enfocados a entornos de procesamiento de datos en lotes que a entornos interactivos.

Cuando este tamaño es muy pequeño, podemos observar que el rendimiento tiende a igualarse independientemente del factor de replicación pero que, según se va incrementando, todos los resultados crecen, pero el factor de replicación favorable crece más. Este fenómeno se ve más claramente en GlusterFS y CephFS, y también en las escrituras de BeeGFS. Esto se sustenta en que, en peticiones de datos tan pequeñas, el *overhead* de contactar con los servicios y escribir los metadatos supera a la mejora de rendimiento que genera el tener un factor de replicación favorable [16]. A partir de un tamaño umbral, ya empieza a ser rentable la cantidad de datos emitida y el rendimiento crece hasta su tope.

En cuanto al tamaño de bloque, si es muy pequeño, significa que los archivos se dividen en muchos bloques minúsculos, generando mayor cantidad de carga de metadatos y mensajes de red, y muchas escrituras pequeñas a los diferentes discos (lo cual puede ser crítico si usamos discos HDD, que necesitarían muchas rotaciones del cabezal), y también de muchas lecturas pequeñas. El más perjudicado por bloques pequeños es HDFS en la escritura, aunque no en la lectura.

Parece mejor tener un tamaño algo más grande de bloque, para establecer menos conexiones cada una de más cantidad de datos. Pero también hay que valorar que si nuestros archivos son muy pequeños, al cargar un bloque solo una parte de los datos obtenidos son los que realmente necesitamos, pudiendo esto lastrar el rendimiento. En nuestra casuística, con archivos de red de cientos de MB, no deberíamos tener ese problema.

Finalmente, el tamaño de archivo. Usar archivos pequeños significa tener más entradas en el espacio de nombres, lo cual vemos que es problemático si, como en HDFS, todos los metadatos se mantienen en memoria [3]. Por otro lado, los archivos pequeños se pueden ver más afectados en sistemas con mucha carga, al tener menos bloques que leer y por lo tanto menos discos que elegir. Con ficheros grandes vemos en todos los sistemas de archivos un rendimiento más constante al producirse una distribución más uniforme de sus diferentes bloques En la cuarta prueba, el rendimiento es bajo, como era esperado emitiendo una cantidad tan pequeña de datos, como podría ser un entorno de lectura de archivos de red donde se hace un pequeña petición cada vez que se captura un paquete.

HDFS en general tiene un comportamiento en la lectura aleatoria muy inferior a sus rivales, ya que está diseñado esencialmente para el acceso secuencial. Una de las razones que también le afectan es el tener que crear muchas conexiones TCP. Además, dado que el tamaño de paquete de datos que se transmite por la red, adecuado para dichos accesos secuenciales, implica tener que transferir muchos datos que van a ser descartados en accesos aleatorios [14].

En BeeGFS podemos comprobar que, en la mayoría de casos, en ambas escrituras el factor

de replicación afecta claramente, siendo las ejecuciones con replicación 2 claramente peores, mientras que en la lectura las velocidades tienden a igualarse en la parte alta. Por lo tanto, se observa que su técnica de Buddy Mirroring es efectiva en mantener el rendimiento en las lecturas, pero no tanto en las escrituras.

CephFS, por otra parte, mantiene una clara diferencia entre el factor de replicación favorable y desfavorable a lo largo de todas las pruebas. El Journaling afecta como cabía esperar, llegando a cortar por la mitad el rendimiento en la escritura, ya que tiene que escribir el doble de datos [11]. Por otro lado, mantiene un rendimiento bastante competitivo en las lecturas.

Con GlusterFS, el hecho de usar el punto de montaje FUSE o `libgfapi` no ha generado tantas diferencias como cabría esperar. Los resultados que obtiene en lecturas aleatorias son claramente peores que los de los demás en los cuatro casos. En los otros tres modos, se aprecia un comportamiento equiparable a los otros DFS usando `libgfapi`, pero el acceso con FUSE, especialmente en escrituras replicadas, deja ver una caída de rendimiento. Se puede apreciar muy claramente en la tercera prueba y en la cuarta. En general, el uso de FUSE hace que el rendimiento caiga en varios casos, dado que el sistema operativo tiene que efectuar más cambios de contexto para pasar las solicitudes de disco del espacio de usuario al de núcleo y viceversa [5].

Por último, en cuanto a Lustre, y teniendo en cuenta que solo se han podido ejecutar las pruebas con un factor de replicación 1, en todos los casos los resultados son estables y con velocidades similares a las de los demás, no viéndose ninguna prueba donde haya una gran caída de rendimiento.

3.4. Conclusión

Primeramente, en el caso de usar replicación de bloques para obtener una buena tolerancia frente a fallos, Lustre aún no es una opción valida, ya que la replicación a nivel de software aún está bajo desarrollo[III]. Esto es especialmente necesario con clusters con hardware de estantería, mientras que en montajes con discos más fiables o donde prime más la velocidad Lustre sería un DFS interesante por su rendimiento estable y cierta facilidad de uso.

Otro aspecto a tener en cuenta es la finalidad. Un típico sistema de Big Data donde los datos se escriben una vez pero se leen muchas se vería favorecido por el uso de BeeGFS, que consigue mantener un rendimiento muy estable en lecturas secuenciales y aleatorias. CephFS también muestra un rendimiento parecido a BeeGFS en lecturas, aunque sufre más con un factor de replicación 1. GlusterFS y HDFS parecen peores opciones ya que, aunque consiguen resultados equiparables a los anteriores en lectura secuencial, en lectura aleatoria se resienten.

Si nos colocamos en el otro extremo, un sistema donde primen las escrituras, habría que valorar si merece la pena invertir dinero en discos SSD para no perder tanto rendimiento en caso de querer usar CephFS. HDFS, con un buen rendimiento con factor de replicación 2, parece una opción viable siempre que no haya un gran número de accesos aleatorios, en tal caso sería mejor BeeGFS o en su defecto GlusterFS. En un sistema de archivos pequeños, los que mejor parecen comportarse con replicación 2 son BeeGFS y Lustre, en escritura, y CephFS y nuevamente BeeGFS en lectura. En el caso de archivos grandes o de tener una variabilidad grande en el tamaño de los archivos, los más estables son BeeGFS y HDFS.

Por último, si se pretendiese usar el sistema de archivos para almacenar paquetes de red en tiempo real, ajustándonos a los resultados de la cuarta prueba, los más adecuados podrían ser HDFS y GlusterFS en las escrituras, y Lustre tanto en escritura como lectura.

En conclusión, para nuestro problema de almacenamiento de paquetes de red, buscaríamos un sistema donde prime la velocidad de lectura, ya que escribiríamos los datos una sola vez y los leeríamos múltiples veces, para operar sobre ellos realizando diferentes análisis de red. Tendríamos un factor de replicación 2 y un tamaño de petición de datos suficientemente grande, ya que ejecutaríamos tareas sobre estos datos en lote. Los ideales parecen ser en tal caso BeeGFS y HDFS.

[III] A fecha 12/05/2018, siendo la última versión estable la 2.11.0.

4 Sistema desarrollado

4.1. Introducción

Hasta ahora hemos comprobado el rendimiento que nos proporcionan los diferentes sistemas de archivos distribuidos con unas pruebas sintéticas, lo cual nos perfila a grandes rasgos su comportamiento. En este capítulo, desarrollaremos un programa de computación distribuida con el afán de enfrentar a cada uno de ellos a un problema real, y ver como se desempeñan.

Para ello usaremos el framework Hadoop de Apache. Apache Hadoop permite el procesamiento distribuido de datos en un cluster como el nuestro. En general, se compone de cuatro partes fundamentales: la librería de utilidades Hadoop Common, el framework de gestión del cluster Hadoop YARN, Hadoop MapReduce, un sistema basado en YARN para procesamiento paralelo y HDFS, que hemos estudiado previamente en la sección 2.2.1, que cumple la función de almacenaje del framework. Una característica interesante es que HDFS se puede desacoplar de Hadoop, permitiendo entonces usar otros sistemas de ficheros, siempre que sus desarrolladores hayan creado un conector, como es nuestro caso y en el de muchos sistemas de archivos distribuidos.

Usaremos esta característica para acoplar nuestros DFS como almacenamiento de Apache Hadoop. Para poder probar el rendimiento en este nuevo escenario, haremos un desarrollo que permitirá ejecutar de manera distribuida un análisis de paquetes de tráfico de red, que se leerán desde nuestros sistemas de ficheros. Este programa que vamos a desarrollar recibirá como entrada paquetes de red almacenados en archivos PCAP y los procesará para analizar sus flujos. Queda fuera del ámbito de este trabajo la captura de dichos paquetes, por lo que supondremos que han sido obtenidos previamente.

Un flujo de red es una sucesión de paquetes de red intercambiados entre dos nodos, tradicionalmente caracterizados por la quíntupla formada por la IP origen, IP destino, puertos origen y destino y tipo de protocolo de transporte usado [17]. Cada paquete será procesado, y finalmente se generará un archivo de texto donde se reflejará una lista de todos los flujos encontrados con algunos valores interesantes relativos al número de paquetes, el tamaño de los mismos y los timestamp. Tomaremos como tiempo de expiración del flujo (tiempo máximo sin encontrar un paquete de dicho flujo [18]) un valor igual a 15 minutos. En la siguiente sección se hará una descripción a más bajo nivel sobre el funcionamiento del software.

El material proporcionado para realizar estas pruebas es una una traza de red de 1000GB y un programa en C llamado `flow_process` que, dado un fichero PCAP de entrada, lee sus paquetes de red y genera el ya comentado archivo de flujos. El `flow_process` realiza un análisis a bajo nivel de las trazas PCAP, leyendo cada paquete que ésta contiene, comprobando si es parte de un flujo ya existente o si debe crear uno nuevo. También lee los demás campos necesarios, como protocolo, timestamp, tamaño, etc, operando a nivel de byte. El archivo de flujos que genera incluye, para cada uno, principalmente la información relativa a timestamp inicio y final del flujo, número de paquetes, tamaño medio del paquete y diferencia temporal entre paquetes, aunque se podría modificar para monitorizar cualquier valor siempre que forme parte de la estructura de un paquete de red.

El programa que queremos desarrollar para poder ejecutar pruebas recibirá una ruta a una carpeta de un sistema distribuido donde se almacenen paquetes de red y los leerá y procesará con `flow_process` de manera distribuida, para finalmente generar un único archivo de flujos

agregado.

4.2. Software desarrollado

Para explicar de manera clara el código desarrollado y las decisiones de diseño, vamos a explicar primero el concepto de MapReduce sobre el que nos basamos, para después pasar a presentar el trabajo desarrollado para dar respuesta a las diferentes problemáticas encontradas. EL programa principal será desarrollado en Java, pero con llamadas a `flow_process`, implementado en C. Con el fin de hacer el uso de estos dos lenguajes compatible, usaremos Java Native Interface (JNI), que nos da la capacidad llamar a funciones de C desde Java. Este framework nos permite declarar una función en Java y su código en C siguiendo este esquema:

```
1     public native String foo(string param);
```

Figura 4.1: Ejemplo de declaración en Java de la función nativa `foo`

```
1     JNIEXPORT jstring JNICALL Java_my_package_MyClass_foo(JNIEnv* env,
          jobject thisObj, jstring param) {
2      /// logica a implementar
3      }
```

Figura 4.2: Implementación en C de la función `foo` según el esquema de JNI

Se puede ver un ejemplo del paso de parámetros en la Figura B.10. El código C ha de ser compilado a una librería de objeto compartido `.so` para que pueda ser cargado por las clases Java que necesiten acceso a ella, como en la figura B.4. Entonces, gracias a JNI, al llamar a una función `native` en Java la Java Virtual Machine (JVM) se encargará de encontrar y ejecutar la implementación presente en la librería.

4.2.1. El paradigma MapReduce

MapReduce fue popularizado por Google [19] como modelo de programación para procesar grandes cantidades de datos de forma sencilla usando computación distribuida en clusters de ordenadores. Consta esencialmente de dos funciones, Map y Reduce. La función Map recibe un par clave-valor y produce otro par clave-valor intermedio. La función Reduce recibe una lista de estos pares, que agrupa todos los valores intermedios creados por Map para una clave, y la reduce a otro valor de salida.

Por lo tanto, el objetivo de este modelo es distribuir la carga de trabajo de entrada entre diferentes nodos, que crean resultados intermedios que son reducidos a un resultado final. En la figura 4.3 se muestra un ejemplo simplificado de funcionamiento para un típico programa WordCount[1]. En dicho esquema están también dos funciones del flujo de ejecución, *splitting* y *shuffling*. En este ejemplo simple, la primera parte un archivo de texto en líneas, y la segunda simplemente agrupa los resultados con igual clave para enviárselos a Reduce. Cabe decir que toda esta lógica la implementará el usuario adaptándolo a sus necesidades.

4.2.2. Funcionamiento del código

Una vez explicado MapReduce con el ejemplo típico de *WordCount* de la figura 4.3, vamos a analizar la implementación de este paradigma que usa Apache Hadoop relacionándola con nuestro código:

entrada y split Los archivos de entrada están previamente almacenados en HDFS o el sistema de ficheros sustituto que estemos usando. El cliente envía un trabajo al `JobTracker`, la clase que gestiona la carga de trabajo, que llama su vez a otra clase `TaskTracker` que se ejecuta en cada nodo. Cada `TaskTracker` inicia su tarea y una clase `InputFormat` lee los archivos de acuerdo a la lógica implementada en un `RecordReader`. En nuestro caso, hemos implementado las clases

[1] `https://wiki.apache.org/hadoop/WordCount`

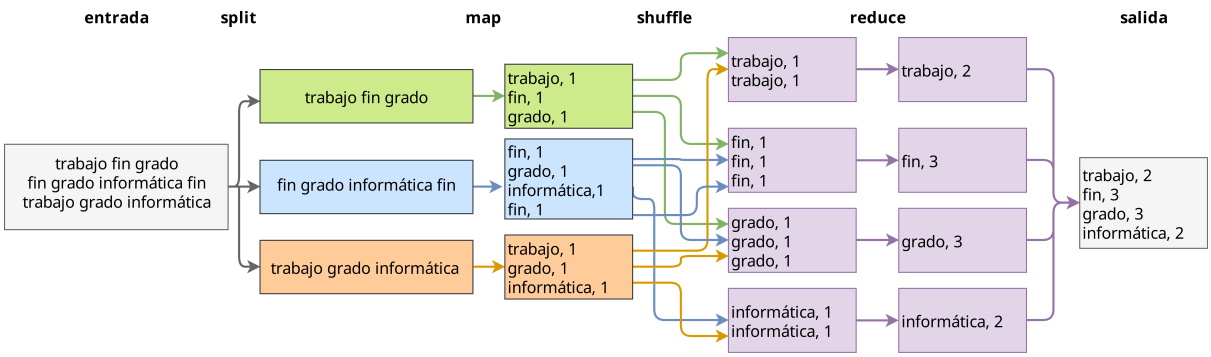


Figura 4.3: Esquema ejemplo del funcionamiento de MapReduce

`PcapFileInputFormat` y `PcapFileRecordReader`, ya que nos encontramos ante la problemática de que el ningún `InputFormat` por defecto está pensado para archivos PCAP, que a diferencia de un archivo de texto plano no se pueden dividir de forma tan simple si queremos procesar los datos que tienen dentro. Para cargar el archivo se usa una clase PCAP, que nos proporciona la librería `pkts.io`[11]. El uso de esta librería es una forma sencilla de tener los PCAP en Java, con la única limitación de que aún no es compatible con el formato más nuevo pcap-ng, solo con el clásico PCAP de *tcpdump*.

map Cada `FileMapper` recibe uno de estos archivos PCAP leídos, y crea un objeto del tipo `CustomPacketHandler`, que es el encargado de realizar la función necesaria con cada una de los paquetes del fichero. Cuando `FileMapper` llama a `loop(pcap)`, `CustomPacketHandler` llamará a su método `nextPacket`, que en nuestra implementación llama al método nativo `process_packet`. La librería nativa mantiene y actualiza una variable global con los diferentes flujos que persiste entre cada llamada. Cuando se acaba la iteración, `nextPacket` devuelve falso y no hay más paquetes en este PCAP, por lo que se llama nuevamente a la librería (`save_flows`) que exporta los valores de dicha variable global de flujos en un archivo local temporal y de ahí `FileMapper` lo copia a HDFS. Finalmente `FileMapper` devuelve una par clave-valor con un 1 y la ruta de este archivo temporal.

El hecho de que cada `FileMapper` lea un PCAP sin partirlo, implica que el número de archivos de entrada será igual al de Mappers. Si los PCAP de entrada son demasiado grandes nos exponemos a perder localidad, ya que cada `FileMapper` deberá leer varios bloques que, cuanto más grande sea el cluster, más probabilidades tendrán de no ser locales a él y estar repartidos arbitrariamente entre los nodos. También, al haber menos tareas ejecutándose, perderíamos granularidad. El caso contrario, PCAP demasiado pequeños, también es un potencial problema dado que, aparte de que el overhead de creación de más JVM es porcentualmente más significante al ser cada `FileMapper` más rápido, si los archivos no ocupan un bloque entero implicaría leer de más para descartar una parte de lo leído. Por lo tanto, lo que parece ideal según esto y teniendo en cuenta los resultados de las pruebas expuestas en el capítulo 3, es tener archivos de tamaño lo más ajustado al tamaño de bloque de cada sistema de archivos, para ni leer datos de más, ni necesitar acceder a múltiples bloques potencialmente foráneos.

shuffle Una clase `SortComparator` ordena por clave los resultados del Mapper. Después, estos datos son enviados por Hypertext Transfer Protocol (HTTP) a los Reducers, por lo que al menos uno de ellos recibirá una lista clave-valor donde todas las claves sean la misma. Es un factor importante de cara a la congestión de red este paso por HTTP [20], ya que habrá que enviar por la red al menos una cantidad de datos igual a la salida de todos los Mappers. En nuestro caso no hemos necesitado crear un `SortComparator` propio.

reduce A continuación, `FileReducer` recibe de los `FileMapper` el par clave-valor con una clave 1 y como valor una lista de rutas de HDFS de los archivos temporales. Cargará desde HDFS todos estos archivos temporales y reducirá los flujos: cuando encuentre dos flujos iguales (que satisfagan

[11] `https://github.com/aboutsip/pkts`

la igualdad de la quíntupla comentada en la sección anterior) los agregará, actualizando las medias y las demás variables. Esto nos asegura que en el archivo final de salida no haya flujos repetidos.

Este último paso además nos da una cierta flexibilidad, ya que el archivo de salida para una misma traza de red será siempre igual, independientemente del número de sub-archivos o del tamaño de los mismos en los que esté partida esa traza, lo que nos da la capacidad de elegir la manera de partir un archivo grande como el nuestro de acuerdo a las características del DFS que estemos usando.

También hay que destacar la ejecución especulativa, una técnica usada en la versión Hadoop de MapReduce para detectar nodos con bajo rendimiento [21]. El trabajo que están efectuando dichas máquinas es duplicado a otras que no tengan tarea en ese momento, intentando reducir así el problema de una máquina lenta marcando el tiempo de ejecución.

En la figura 4.4 se presenta un esquema simplificado de ejecución del código (en rojo los pares clave-valor pasados, en amarillo las funciones `native` que tienen su implementación en C). Todo este código lo hemos compilado con maven en un fat jar, un Java ARchive (JAR) que incluye dentro todas las dependencias necesarias para su ejecución.

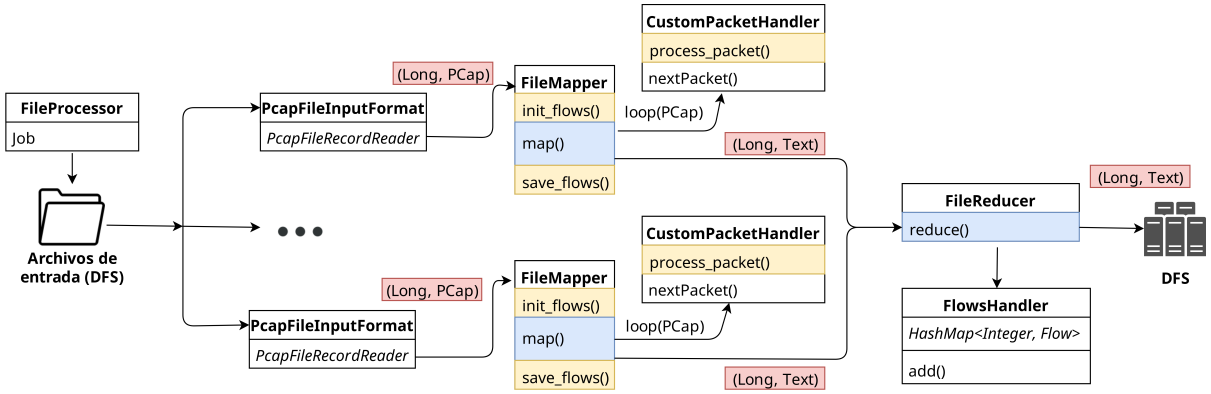


Figura 4.4: Diagrama de ejecución de nuestro programa en MapReduce

4.3. Pruebas

4.3.1. Configuración de las pruebas

Una vez finalizado el código, pasaremos a realizar pruebas con él. Para esta ocasión, hemos elegido el caso favorable y más realista, es decir, con factor de replicación 2 y las optimizaciones de las pruebas del capítulo 3. Por lo tanto, Lustre no entrará en esta prueba al no disponer de replicación a nivel de software. En el cuadro 4.1 se listan los parámetros de uso de memoria que hemos configurado de acuerdo a nuestros nodos para que el programa funcione correctamente sin que ninguna tarea se quede sin memoria.

En cuanto a la configuración de los conectores, con GlusterFS[III] y BeeGFS[IV] hemos instalado el JAR del conector oficial en la carpeta `/usr/local/hadoop/share/hadoop/common` así como dentro del ejecutable. En el caso de CephFS hemos necesitado compilar manualmente el código de la última version desde su GitHub[V], dado que la proporcionada en la página oficial solo funciona con Hadoop 1.

Una vez copiado el conector, el cambio de sistema de ficheros se ejecuta en dos partes. Primero, se configuran las opciones `fs.AbstractFileSystem` y `fs.impl` del archivo `core-site.xml`, indicando de que clase será nuestro sistema de archivos nuevo, junto a otras opciones que cada desarrollador haya considerado. Después, hay que incluir en el Classpath de Hadoop la ruta al conector con la implementación de la interfaz `AbstractFileSystem`, que incluye métodos para

[III] `https://github.com/gluster/glusterfs-hadoop`
[IV] `https://www.beegfs.io/wiki/HadoopConnector`
[V] `https://github.com/ceph/cephfs-hadoop`

Cuadro 4.1: Parámetros de Yarn y MapReduce (`yarn-site.xml` y `mapred-site.xml`)

Nombre de la propiedad	**Descripción**	**Valor usado**	**Valor por defecto**
`yarn.nodemanager.resource.memory-mb`	Máxima RAM usable en nodo	`58000`	`8192`
`yarn.nodemanager.resource.cpu-vcores`	Máximo número de Cores usable en nodo	`24`	`8`
`yarn.scheduler.maximum-allocation-mb`	Valor de incremento de las peticiones de memoria	`12288`	`8192`
`mapreduce.map.memory.mb`	Máxima RAM usable (Mapper)	`4096`	`1024`
`mapreduce.map.java.opts`	Máxima RAM usable por la JVM de un Mapper	`-Xmx3072m`	`-Xmx512m`
`mapreduce.reduce.memory.mb`	Máxima RAM usable (Reducer)	`12288`	`1024`
`mapreduce.reduce.java.opts`	Máxima RAM usable por la JVM de un Reducer	`-Xmx8192m`	`-Xmx512m`

abrir y leer archivos, crear carpetas, y otras utilidades. Por ejemplo, si tenemos un sistema montado con GlusterFS, Apache Hadoop leerá de los archivos de configuración que la clase del DFS es `org.apache.hadoop.fs.glusterfs.GlusterFileSystem`, y buscará en sus JAR una implementación de dicha clase para poder efectuar la conexión. En la imagen 4.5 se ilustra el uso de BeeGFS como sustituto de HDFS: podemos usar el punto de montaje de BeeGFS de la manera habitual, pero además acceder a los datos mediante la interfaz de Hadoop.

```
[root@neon jorgecf]# hadoop fs -ls -h /user/onepcap
Found 1 items
-rw-r--r--   1 root root    542.5 M 2018-05-02 18:31 /user/onepcap/final_00440_20180309171546.pcap
[root@neon jorgecf]# ll /mnt/beegfs/hadoop/user/onepcap/ -h
total 543M
-rw-r--r--. 1 root root 543M May  2 18:31 final_00440_20180309171546.pcap
```

Figura 4.5: Imagen ilustrativa del uso de BeeGFS con Hadoop

Hemos elegido 512MB como tamaño de archivo de acuerdo a los resultados del capítulo 3, y teniendo en mente que se cargará entero en memoria, por lo que no nos conviene un tamaño demasiado grande que reduzca el número de Mappers ejecutándose concurrentemente. Como tamaño de bloque se ha configurado el mismo valor, viendo que tiene un rendimiento aceptable en los resultados de todos los DFS del capítulo 3, y contando con lo comentado en la sección anterior sobre la relación de estos dos parámetros.

En cuanto a los archivos de entrada, para poder partir la traza de 1000GB en archivos de 512MB, hemos tenido que usar software adicional, `capinfos`[VI] y `editcap`[VII]. Con `capinfos` calculamos la longitud media de cada paquete en dicha traza, como se ve en la figura 4.6, y con `editcap` hemos partido el archivo en trozos, cada uno con 382.312 paquetes (el resultado de dividir 512MB entre 1339.22 bytes/paquete). Podemos calcular el número de paquetes de red que vamos a tener que procesar, aproximadamente 747 millones.

Figura 4.6: Imagen con el resultado del análisis de `capinfos`

```
[root@neon jorgecf]# cat cap.info
File name:           doclab.pcap
Average packet size: 1339.22 bytes
```

Para obtener nuestras métricas de rendimiento, hemos usado los `Counters` que nos proporciona Hadoop, excepto para el tiempo de ejecución, que ha sido obtenido por nosotros (figura B.2). Estos elementos son variables globales que mantiene y actualiza el framework para monitorizar ciertos campos, y que se pueden acceder a ellos de manera individual para cada trabajo. En la tabla 4.2 se describen cuáles hemos utilizado. El método de acceso se puede ver en la figura B.8.

[VI] `https://www.wireshark.org/docs/man-pages/capinfos.html`
[VII] `https://www.wireshark.org/docs/man-pages/editcap.html`

Cuadro 4.2: Lista de `Counters` de Hadoop empleados para las gráficas

Nombre del contador	Descripción
`MILLIS_MAPS`	Suma del tiempo de ejecución de todas las tareas de Map
`MILLIS_REDUCES`	Suma del tiempo de ejecución de todas las tareas de Reduce
`MAP_INPUT_RECORDS`	Número de `Input Splits` de entrada al Job (igual al número de archivos)
`MAP_OUTPUT_BYTES`	Suma del tamaño de la salida generada por las tareas de Map (y por tanto tamaño de entrada a Reduce)
`VIRTUAL_MEMORY_BYTES`	Suma de memoria RAM y Swap utilizada por todas las tareas
`PHYSICAL_MEMORY_BYTES`	Cantidad de memoria RAM utilizada por todas las tareas
`TOTAL_LAUNCHED_TASKS`	Número total de tareas lanzadas

Primera ejecución y optimizaciones

Para hacer una primera prueba del código desarrollado, cogimos una muestra de 55GB en archivos de 512MB, con la que vemos que el rendimiento no es el esperado. Primero comprobamos que el reductor se ejecuta bastante rápido hasta el 70 %, pero a partir de ahí su rendimiento cae, incluso impidiendo que el programa termine. Esto parece indicar que, cuando el `ArrayList` interno de `FlowsHandler` está muy lleno, el rendimiento del mismo es no es bueno. Según la implementación de `ArrayList`, el método `indexOf()` que estábamos usando itera sobre toda la lista hasta que encuentra un resultado. Como solución cambiamos a un `HashMap`, cuyo método `get()` nos resulta más eficiente.

El siguiente problema que tenemos es en el `toString()` de la clase `FlowsHandler`. Comprobamos que para imprimir al archivo de resultados, es demasiado lento e igualmente evita que el trabajo finalice. Nuestro fallo consiste en usar la concatenación de string dentro de un bucle de `FlowsHandler` que itera sobre todos los `Flow`. Los string no son mutables, y en cada iteración se crea un nuevo string [22, pg. 506], haciendo un gasto enorme de memoria. Lo sustituimos por un `StringBuilder`, que nos da un rendimiento mejor.

Con esta primera ejecución también comprobamos que, para los 55GB, tanto `flow_process` como el software desarrollado generaban el mismo archivo final de flujos (únicamente era diferente el id interno, los valores se mantenían iguales).

También hemos tenido en cuenta el parámetro `mapred.job.reuse.jvm.num.tasks`, establecido en `mapred-site.xml`. Define cuantas tareas de Map o de Reduce ejecutará cada JVM, su valor por defecto es 1. En tareas muy largas en tiempo, esta configuración parece funcionar bien, pero con tareas tan cortas como las nuestras (unos pocos segundos) el overhead de crear y destruir una instancia de JVM supone una parte significante del tiempo total [23, 24]. Elegimos un valor de 3, de acuerdo a los resultados de la figura 4.7. Como podemos observar, el tiempo agregado de todas las tareas (la suma del tiempo que han tardado en ejecutar todos los Maps y Reduces, en otras palabras) se mantiene muy constante, mientras que el tiempo de ejecución efectivo (desde que se inicia el Job hasta que escribe los resultados y finaliza) se reduce un 36 % entre reuse con valor de 1 y 3. Esto apunta que, mientras que las tareas una vez iniciada la JVM tardan lo mismo en terminar (lo cual es esperable), la reducción a un tercio de creaciones de ésta, mejora el tiempo de ejecución.

4.3.2. Análisis de resultados

Vamos a comenzar enfocando la inspección de los resultados al tiempo de ejecución. Primeramente, en la figura 4.8 tenemos los resultados de tiempo de ejecución en formato de tiempo a la izquierda y normalizados respecto a HDFS a la derecha.

Comprobamos que el más rápido en ejecutar es BeeGFS, que es capaz de procesar nuestro terabyte de datos en treinta y un minutos (`532.77mbps` o `398.000` paquetes por segundo), seguido de cerca por HDFS. Aunque como es esperable no se acerca al rendimiento del capítulo 3, donde las pruebas solo realizaban peticiones de lectura/escritura, este resultado es interesante teniendo

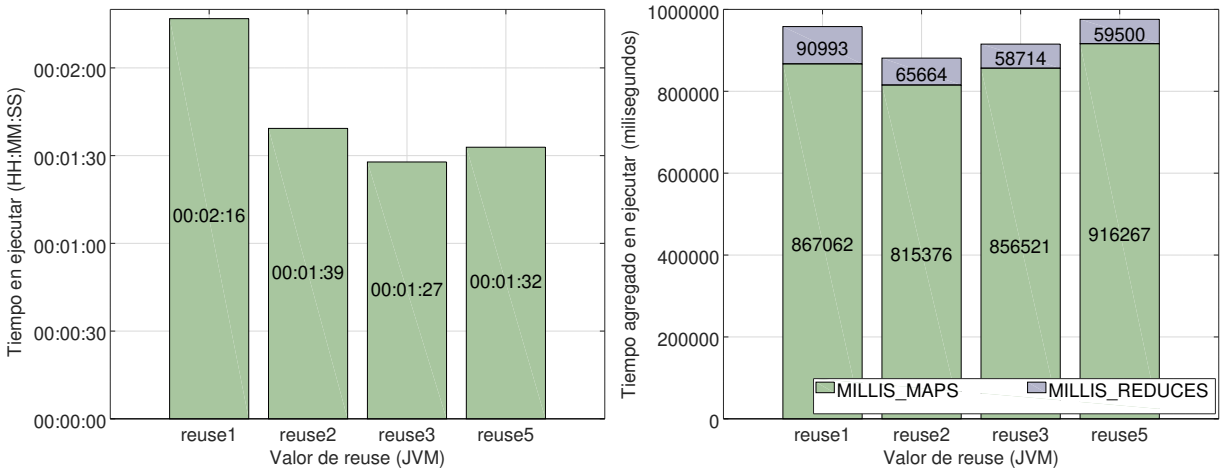


Figura 4.7: Gráficas de tiempo de ejecución respecto al valor de `mapred.job.reuse.jvm.num.tasks`

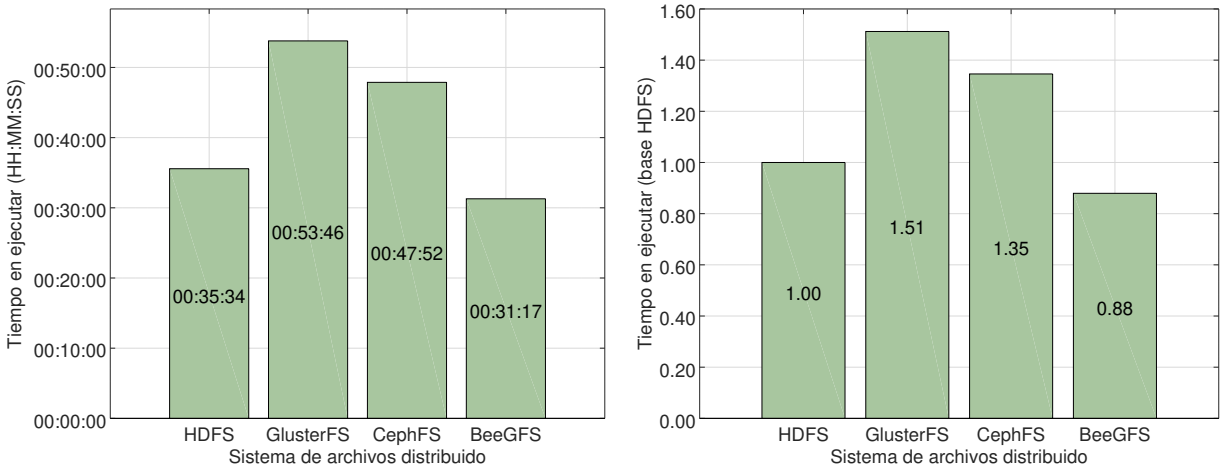


Figura 4.8: Gráficas de tiempo de ejecución con MapReduce

en cuenta el *overhead* añadido por al framework de Hadoop y el propio procesamiento a bajo nivel de todos y cada uno de los paquetes de red.

En cuanto a nomenclatura, hay que distinguir entre la fase de Map, que incluye el paso de los datos a los Mappers, la propia ejecución del Map, el sorting de resultados y el shuffling, con la ejecución de la clase `FileMapper` que implementa la interfaz `Mapper`. En la gráfica 4.9 observamos la velocidad de procesamiento media, es decir, dividiendo los 1000GB de entrada que reciben los Mappers entre el número de ellos, y la salida de los Mappers entre el número de Reducers, para la fase de Reduce. Podemos apreciar valores muy similares excepto en CephFS, un 47 % más rápido en la fase de Map que el siguiente peor, BeeGFS. Por lo que vemos en esta gráfica, aunque las fases de CephFS se ejecutan más velozmente, en velocidad de ejecución total consigue peor rendimiento. Esto apunta a que con CephFS

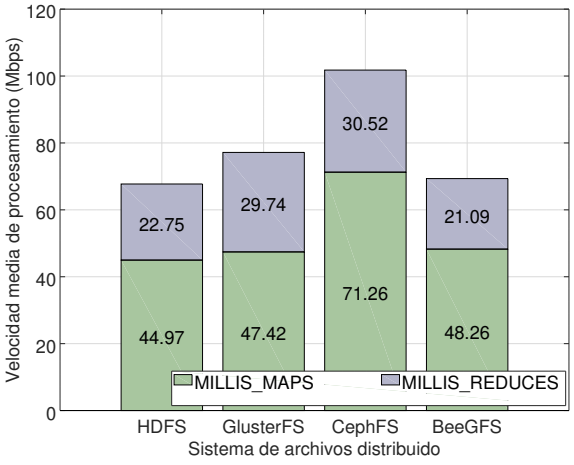


Figura 4.9: Gráfica de velocidad media de procesamiento con MapReduce

se pierde paralelismo y se consiguen ejecutar
menos tareas a la vez, por lo que hemos monitorizado el uso de memoria para investigar esto más a fondo.

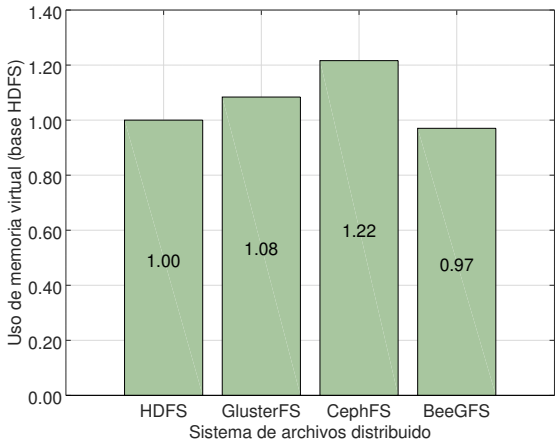


Figura 4.10: Gráfica con el uso agregado de memoria virtual con MapReduce (normalizado a 1)

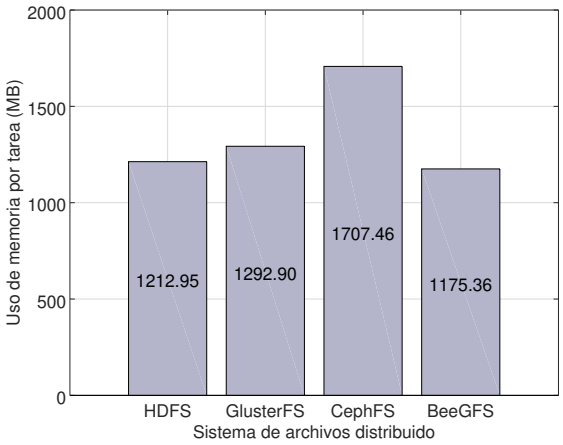


Figura 4.11: Gráfica con el uso de memoria RAM por tarea con MapReduce

Primero, en la gráfica 4.10 tenemos el uso de memoria virtual agregado y normalizado en base a HDFS. BeeGFS, el más rápido, es también el único que usa menos memoria que HDFS, que está a un nivel cercano a GlusterFS. Además, en la figura 4.11 tenemos el uso medio de memoria física por cada tarea. Observamos claramente como CephFS tiene un gasto de RAM mayor, un 40 % más que HDFS. En tareas típicas de Big Data de procesamiento de terabytes de datos, debemos valorar si disponemos de un cluster austero o de uno más potente, de terabytes de RAM, que pueda cargar todos los archivos y ejecutar sobre ellos todas las tareas de manera paralela. Nuestro caso sería el primero, donde el tamaño de la traza a procesar es mucho mayor a la memoria física del cluster.

Es por esto por lo que estamos ante un *trade-off* velocidad-uso de memoria: queremos un sistema de archivos que ejecute rápido las peticiones que se le envían, pero que además use la menor memoria posible para dar cabida al número máximo de ejecuciones concurrentes. Parece claro que, aunque CephFS sea más veloz, malogra la ventaja conseguida como consecuencia de un gasto mucho mayor de recursos. El caso de GlusterFS parece ser diferente, y aunque el uso de memoria está a la par de BeeGFS y HDFS, su tiempo de ejecución es peor. Esto podría venir por la degradación de rendimiento al usar FUSE sobre una cantidad tan grande de datos.

Como broche final, hemos ejecutado `flow_process` como programa independiente sobre la traza entera, para comprobar la mejora de rendimiento obtenida al distribuir los datos y el procesamiento. Hemos obtenido que el tiempo total de ejecución son `222` minutos, a `75.08 mbps`. Por tanto, el uso del sistema de almacenamiento distribuido BeeGFS para almacenar los datos nos ha proporcionado una reducción del tiempo de ejecución en un `85.91 %`, es decir, más de siete veces menos tiempo de ejecución.

4.4. Conclusión

En este capítulo hemos partido de un código C qué, gracias a JNI, hemos conseguido integrar con las librerías de MapReduce de Apache Hadoop para usarlo de manera distribuida en nuestro propio trabajo orientado a archivos de red. Después hemos presentado el rendimiento en varios planos, viendo que los dos mejores sistemas de archivos distribuidos para nuestro caso concreto son, tal y como parecían pronosticar los resultados del capítulo 3, HDFS y BeeGFS. GlusterFS y CephFS se quedan atrás, uno por el uso de FUSE y otro por un uso elevado de recursos, algo que ya habíamos experimentado debido al Journaling.

5 Conclusiones y trabajo futuro

5.1. Conclusiones

En este trabajo de fin de grado, hemos comenzado planteando el problema del almacenamiento y procesamiento de paquetes de tráfico de red de manera distribuida y eficiente, y desde esa base hemos realizado un estudio comparativo de diferentes sistemas de archivos distribuidos.

En el capítulo 2 se han explicado los conceptos centrales de cualquier sistema de archivos distribuido, para después detallar de qué manera estos conceptos son implementados en los cinco DFS que hemos estudiado.

Lo más importante se encuentra en los capítulos 3 y 4. En el primero, hemos buscado encontrar las diferencias de rendimiento usándolos únicamente como sistema de almacenamiento. Se han encontrado resultados similares entre todos los DFS, pero con ciertos matices. Hemos observado que HDFS es un sistema bastante peor en accesos aleatorios mientras que BeeGFS perdía poco rendimiento. También hemos encontrado en Lustre un software que, a pesar de estar algo incompleto por la falta de replicación a nivel de archivo, mantiene comportamientos muy estables y predecibles. Lo mismo puede decirse de CephFS, con el pero de un rendimiento en escritura a un nivel más bajo por su Journaling. En GlusterFS, hemos encontrado diferencias bastante altas en algunos casos entre el uso de su librería `libgfapi` y el acceso por punto de montaje FUSE, de cuya estructura se habló en la sección 2.1.2. Como se habla en la conclusión de este capítulo 3, orientándonos al problema de almacenamiento de paquetes de red necesitaríamos una solución capaz de aportar un buen comportamiento con paquetes de tamaño mediano, un escenario de transmisión continua de datos de manera secuencial así como también un rendimiento mejor en la lectura. En ese caso los candidatos ideales como sistema de archivos distribuido son BeeGFS y HDFS, por ser los que mejor se portan en dichas características.

Para el capítulo 4, hemos escogido un sistema de procesamiento muy usado como es Apache Hadoop, y hemos integrado en él nuestros DFS. De esta manera, hemos podido probar los sistemas de almacenamiento como componente de un sistema de procesamiento distribuido. Esto ha implicado una programación en Java así como también el uso de la librería de Java JNI para poder usar en este código el programa en C nativo flow_process. Con esto hemos conseguido ejecutar un programa de análisis de paquetes a muy bajo nivel de manera repartida entre varios nodos, de una manera eficiente y fiable.

Los resultados que hemos obtenido ejecutando esta tarea se pueden ver en la sección 4.3.2, donde hemos introducido el *trade-off* entre uso de RAM y velocidad en gestionar las peticiones de datos. Esto se ve en CephFS, ya que aunque es más rápido en terminar sus tareas (figura 4.9) el hecho de usar bastante más memoria merma el paralelismo y por tanto el rendimiento final (figuras 4.8 y 4.11). GlusterFS ha sido el peor de la prueba, por su uso de FUSE, mientras que el mejor ha sido BeeGFS, que ha conseguido procesar un terabyte de datos en media hora (532.77mbps), seguido de HDFS.

Como conclusión, BeeGFS sería el candidato ideal en entornos distribuidos para el almacenamiento y procesamiento de datos de red.

5.2. Trabajo futuro

La posible investigación futura, siguiendo la estela de este trabajo, se podría abordar de diferentes maneras. Primero, en lo tocante al capítulo 2, cabría valorar extenderlo con nuevos

sistemas distribuidos que están en boga, como MooseFS[I], con una arquitectura similar a Lustre y BeeGFS, XtreemFS[II], que asegura tener una excepcional tolerancia frente a fallos, o Quobyte [III]. Con esto podríamos hacer un análisis de mercado mucho más extenso del que cabe realizar en un trabajo de fin de grado.

En cuanto al capítulo 3, resultaría interesante agrandar nuestro cluster en dos aspectos. Por un lado, expandir el número de nodos para analizar la relación entre el tamaño del cluster y el rendimiento, y ver de qué manera escala sus resultados cada sistema de archivos. También expandirlo para crear un cluster heterógeneo, con discos de diferentes velocidades y calidades, con el fin de comprobar si el software es suficientemente inteligente como para repartir menos carga de trabajo a discos más lentos, y así evitar que estos marquen la velocidad. Lógicamente, para llevarlo a cabo se necesitaría una inversión adicional en hardware.

Una vez ampliado nuestro cluster, y más relacionado con el capítulo 4, se podrían realizar ciertas modificaciones no triviales al código desarrollado. En primer lugar, para evitar tener que partir manualmente el archivo de 1000GB, añadir algún tipo de preprocesamiento que divida el archivo en paquetes del tamaño dado y los copie al DFS. Con la librería `pkts.io` podríamos hacer esto, además de partir el archivo en partes exactas de, por ejemplo, 512MB ya que nos da el tamaño de cada paquete en concreto, al contrario que `capinfos` que calcula la media. Una posible opción sería hacer este preprocesamiento antes de la ejecución del trabajo[IV] introduciendo conceptos nuevos como `SequenceFile`.

[I] `https://moosefs.com/`
[II] `http://www.xtreemfs.org/`
[III] `https://www.quobyte.com/`
[IV] `https://github.com/marouni/pcap2seq`

Bibliografía

[1] Michael Kende: *Global Internet Report 2016*, 2016.

[2] Kuai Xu, Zhi Li Zhang y Supratik Bhattacharyya: *Internet traffic behavior profiling for network security monitoring.* IEEE/ACM Transactions On Networking, 16(6):1241–1252, 2008.

[3] Dhruba Borthakur: *The Hadoop Distributed File System: Architecture and design*, 2008.

[4] Sarp Oral, Feiyi Wang, David Dillow, Galen M Shipman, Ross Miller y Oleg Drokin: *Efficient Object Storage Journaling in a Distributed Parallel File System.* En *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST10)*, páginas 1–12. USENIX, 2010.

[5] Sophal Hong: *Eliminating FUSE Bottleneck by Implementing Multi-threaded Framework on Distributed File System.* Tesis de Doctorado, Escuela de Ingeniería, Universidad Nacional de Seúl, Agosto 2017.

[6] Bharath Kumar Reddy Vangoor, Vasily Tarasov, Erez Zadok: *To FUSE or Not to FUSE: Performance of User-Space File Systems.* En *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST17)*. USENIX, Marzo 2017.

[7] Wiki, GlusterFS. `https://docs.gluster.org/en/v3/Administrator%20Guide`. [Online; accedido el 19/05/2018].

[8] Frank Herold, Sven Breuner y Jan Heichler: *An introduction to BeeGFS*, Marzo 2018.

[9] Wiki, CephFS: *CephFS Architecture.* `http://docs.ceph.com/docs/master/architecture/`. [Online; accedido el 20/05/2018].

[10] Wiki, Lustre: *Introduction to Lustre Architecture.* `http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf`, Octubre 2017.

[11] Feiyi Wang, Mark Nelson, Sarp Oral, Scott Atchley, Sage Weil, Bradley W. Settlemyer, Blake Caldwell y Jason Hill: *Performance and scalability evaluation of the Ceph parallel file system.* En *Proceedings of the 8th Parallel Data Storage Workshop*, página 18. ACM, 2013.

[12] Sanam Shahla Rizvi, Tae Sun Chung: *Flash SSD vs HDD: High performance oriented modern embedded and multimedia storage systems.* En *2nd International Conference on Computer Engineering and Technology (ICCET 2010)*, volumen 7, páginas V7–298. IEEE, 2010.

[13] Subhash Saini, Jason Rappleye, Johnny Chang, David Barker, Piyush Mehrotra y Rupak Biswas: *I/O performance characterization of Lustre and NASA applications on Pleiades.* En *19th International Conference on High Performance Computing (HiPC 2012)*, páginas 1–10. IEEE, 2012.

[14] Wei Zhou, Jizhong Han, Zhang Zhang y Jiao Dai: *Dynamic Random Access for Hadoop Distributed File System.* En *32nd International Conference on Distributed Computing Systems Workshops (ICDCSW 2012)*, páginas 17–22. IEEE, 2012.

[15] Cairong Yan, Tie Li, Yongfeng Huang y Yanglan Gan: *Hmfs: Efficient Support of Small Files processing over HDFS.* En *International Conference on Algorithms and Architectures for Parallel Processing*, páginas 54–67. Springer, 2014.

[16] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel y Thomas Ludwig: *Small-file access in parallel file systems.* En *IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009)*, páginas 1–11. IEEE, 2009.

[17] Shane Amante, Jarno Rajahalme, Brian E. Carpenter y Sheng Jiang: *IPv6 Flow Label Specification.* RFC 6437, Noviembre 2011. `https://tools.ietf.org/html/rfc6437`.

[18] Benoit Claise: *Cisco Systems NetFlow Services Export Version 9.* RFC 3954, Octubre 2004. `https://tools.ietf.org/html/rfc3954`.

[19] Jeff Dean, Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters.* En *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI04)*, página 10. USENIX, 2004.

[20] Eric Sammer: *Hadoop Operations*, capítulo 4, páginas 68–69. O'Reilly Media, Inc., 1ª edición, Septiembre 2012.

[21] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz y Ion Stoica: *Improving MapReduce performance in heterogeneous environments.* En *8th USENIX Symposium on Operating Systems Design and Implementation*, número 4, página 7. USENIX, 2008.

[22] James Gosling, Bill Joy, Guy Steele, Gilad Bracha y Alex Buckley: *The Java Language Specification*, Julio 2013.

[23] Tom White: *Hadoop: The definitive guide*, capítulo 6, páginas 219–220. O'Reilly Media, Inc., 3ª edición, 2012.

[24] Robert Stewart y Jeremy Singer: *Comparing fork/join and MapReduce.* Cite-seer, Tech. Rep., páginas 14–16, 2012.

[25] Intel: *Tuning Lustre Systems.* `https://www.intel.com/content/dam/www/public/us/en/documents/presentation/tuning-lustre-systems-training.pdf`. [Online; accedido el 2/05/2018].

[26] Wiki, BeeGFS: *Tips and Recommendations for Storage Server Tuning.* `https://www.beegfs.io/wiki/StorageServerTuning`, Febrero 2018. [Online; accedido el 2/05/2018].

[27] Blog, Ceph: *Ceph Bobtail Performance – IO Scheduler Comparison.* `https://ceph.com/community/ceph-bobtail-performance-io-scheduler-comparison`, Enero 2013. [Online; accedido el 2/05/2018].

[28] Docs, Gluster: *Linux kernel tuning for GlusterFS.* `https://docs.gluster.org/en/v3/Administrator%20Guide/Linux%20Kernel%20Tuning/`. [Online; accedido el 2/05/2018].

[29] AMD: *Hadoop Tuning Guide.* `https://developer.amd.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf`, Octubre 2012. [Online; accedido el 2/05/2018].

[30] DP Traynor, TS Froy y CJ Walker: *Upgrading and Expanding Lustre Storage for use with the WLCG.* En *Journal of Physics: Conference Series*, volumen 898, página 4. IOP Publishing, 2017.

A

Optimizaciones de los sistemas de archivos

A.1. Ajustes genéricos

Estos primeros ajustes son sobre el propio kernel y el sistema operativo:

I/O Scheduler Es la lógica que usa el sistema operativo para decidir el orden en el que las peticiones de I/O de los programas serán remitidas a los discos físicos. Las soportadas por defecto en nuestro CentOS son deadline, Completely Fair Queuing (CFQ) y noop, todas ellas diseñadas por Jens Axboe.

Deadline mantiene dos colas, una ordenada y otra de deadline, intentando siempre evitar que los trabajos superen su tiempo establecido de ejecución. CFQ crea varias colas donde aloja las peticiones enviadas por los procesos, y reserva una porción de tiempo a cada elemento encolado. Noop es el planificador más sencillo, se basa en una cola First In First Out (FIFO) donde se almacenan las peticiones.

El planificador deadline parece funcionar bien sobre todos los sistemas de archivos[25, 26, 27, 28], excepto sobre HDFS, donde nos decantamos por CFQ[29].

Hacemos el ajuste con una de las dos siguientes líneas en cada máquina:

```
echo deadline > /sys/block/sd{b..h}/queue/scheduler
echo cfq > /sys/block/sd{b..h}/queue/scheduler
```

Queue Otros ajustes sobre el kernel que hemos usado son[30]:

```
echo 4096 > /sys/block/sd{b..h}/queue/nr_requests
```

Hemos aumentado el tamaño de la cola de peticiones del planificador. Con esto debería aumentar su margen para elegir la siguiente petición.

```
echo 4096 > /sys/block/sd{b..h}/queue/read_ahead_kb
```

Aumentamos la anticipación de lectura. Es la cantidad de bytes que el kernel va a leer por adelantado para reducir el tiempo que un proceso está esperando a tener datos disponibles.

A.2. HDFS

En las opciones genéricas hemos introducido los siguientes flags de la JVM:

```
-XX:+AggressiveOpts -XX:+UseCompressedOops
-XX:+UseBiasedLocking  -XX:+OptimizeStringConcat
```

AgressiveOpts ajusta un grupo de optimizaciones incluidas en versiones más recientes. UseCompressedOops comprime los puntero de 64 bit para reducir el uso de memoria. UseBiasedLocking comprueba si un elemento bloqueado por un hilo no está siendo usado en ningún otro, y ejecuta las operaciones sobre dicho elemento sin incurrir en costes de sincronización. OptimizeStringConcat intenta optimizar la creación de Strings mediante concatenaciones.

Y en las opciones del NameNode las siguientes:

```
-XX:+UseConcMarkSweepGC -XX:ParallelGCThreads=8
-Xms1G -Xmx1G
-XX:+UseFastAccessorMethods
```

Las dos primeras opciones buscan optimizar el Garbage Collector. Los dos siguientes establecen en 1G la memoria por defecto y máxima del JVM.

A.3. GlusterFS

Hemos reconfigurado las siguientes opciones de Gluster:

Aumentamos el tamaño máximo para cachear un archivo:

```
performance.cache-max-file-size: 64MB
```

Aumentamos igualmente el tamaño de la caché:

```
performance.cache-size: 8GB
```

Esta opción optimiza la búsqueda en en la DHT:

```
cluster.lookup-optimize: on
```

A.4. BeeGFS

En el archivo de configuración beegfs-storage.conf subimos el numero de hilos (workers) y el tamaño de la lectura por adelantado:

```
tuneNumWorkers                = 10
tuneFileReadAheadSize         = 4m
```

Y en el archivo beegfs-client.conf, subimos el número máximo de conexiones del cliente al servidor:

```
connMaxInternodeNum            = 16
```

A.5. CephFS

Las optimizaciones qué han ido especialmente enfocadas a aliviar el problema del Journaling. Las dos primeras opciones permiten hacer I/O directa en el Journaling. La tercera establece el uso de escrituras asíncronas con libaio:

```
journal dio = true
journal block align = true
journal aio = true
```

Subimos el tamaño efectivo del Journal:

```
journal_max_write_entries = 5000
journal_max_write_bytes = 1048576000
```

Aumentamos el número máximo de operaciones y tamaño de la cola del Journal:

```
journal_queueu_max_ops = 3000
journal_queue_max_bytes = 1048576000
```

En cuanto a las demás optimizaciones, primero subimos el tamaño de la cola general de archivos:

```
filestore_queue_max_ops=5000
filestore_queue_max_bytes = 1048576000
```

Intervalo máximo en el que se hace flush de los datos a disco:

```
filestore_max_sync_interval = 10
```

A.6. Lustre

Con el uso de LNet ajustamos algunos parámetros de red del sistema operativo:

Aumentamos el valor por defecto de los buffers de envío y recepción para cualquier tipo de conexión. El valor que elegimos es el mismo que wmem_max y rmem_max (el valor máximo de dichos buffers):

```
echo 4194304 > /proc/sys/net/core/wmem_default
echo 4194304 > /proc/sys/net/core/rmem_default
```

En cuanto a parámetros propios del software:

Al seleccionar el tamaño de bloque, elegimos el stripe-index -1 en vez de 0. Esta propiedad controla sobre que número de disco se empieza a escribir o leer. Con -1, dejamos que Lustre elija el que quiera mediante su Round-Robin, ya que elegir 0 u otro en concreto puede producir cierto embotellamiento si se lanzan muchas tareas concurrentemente:

```
lfs setstripe --stripe-size 8M --stripe-index -1 --stripe-count -1 /mnt/lustre
```

Aumentamos también el tamaño de caché máximo en el cliente:

```
lctl set_param llite.*.max_cached_mb=32768
```

Y por último doblamos la cantidad de MB “sucios”:

```
lctl set_param osc.*.max_dirty_mb=64
```

B

Código desarrollado

A continuación se incluye el código Java desarrollado, cuya funcionalidad se describe en el capítulo cuarto. Se han obviado algunos constructores y métodos toString para reducir espacio.

```
1  /**
2   * Lee un archivo PCap para que FileMapper pueda usarlo.
3   *
4   * @author Jorge Cifuentes <jorge.cifuentes95@gmail.com>,
        <jorge.cifuentesf@estudiante.uam.es>
5   * @date 12 apr 2018
6   */
7  public class PcapFileInputFormat extends FileInputFormat<LongWritable, Pcap> {
8
9    @Override
10   protected boolean isSplitable(JobContext context, Path file) {
11     return false;
12   }
13
14   @Override
15   public RecordReader<LongWritable, Pcap> createRecordReader(InputSplit split,
         TaskAttemptContext context) {
16     return new PcapFileRecordReader();
17   }
18 }
```

Figura B.1: Código fuente de `tfg.Pcap.PcapFileInputFormat`

```java
/**
 * Esta clase se ocupa de instanciar un Job de Apache Hadoop y
 * guardar los resultados en un archivo local.
 *
 * @author Jorge Cifuentes <jorge.cifuentes95@gmail.com>,
    <jorge.cifuentesf@estudiante.uam.es>
 * @date 26 apr 2018
 */
public class FileProcessor {

    public static void main(String[] args) throws Exception {

        if (args.length < 2) {
            System.out.println("[ERROR] necesito la carpeta de entrada del fs
                y el nombre del csv de salida como parametro");
            System.exit(0);
        }

        /* Parametros pasados */
        String inputFolder = args[0];
        String outputFile = System.getProperty("user.dir") + "/" + args[1];

        String date = new SimpleDateFormat("dd_MM_yyyy_hh_mm_ss").format(new
            Date());

        Configuration conf = new Configuration();

        Job job = new Job(conf, "FileProcessor");
        job.setJarByClass(FileProcessor.class);


        /* Nuestras clases mapeadoras y reductoras */
        job.setMapperClass(FileMapper.class);
        job.setReducerClass(FileReducer.class);

        /* Clases de key-value que se pintaran en el archivo de salida que
            genere el reductor */
        job.setOutputKeyClass(LongWritable.class);
        job.setOutputValueClass(Text.class);

        /* Tipo de archivo de entrada y archivo de salida de nuestro trabajo */
        job.setInputFormatClass(PcapFileInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        /* Rutas de entrada y salida */
        FileInputFormat.addInputPath(job, new Path("/" + inputFolder + "/"));
        FileOutputFormat.setOutputPath(job, new Path("/mapredjobs_" + date +
            "/"));

        /* EJecutamos el trabajo */
        long start = new Date().getTime();
        boolean result = job.waitForCompletion(true);
        long end = new Date().getTime();

        System.out.println("Resultado: " + result);

        /* Escribimos los resultados de este trabajo */
        JobResultsWriter jrw = new JobResultsWriter(outputFile, job, (end -
            start));
        jrw.write();
    }
}
```

Figura B.2: Código fuente de `tfg.FileProcessor`

```java
/**
 * Este RecordReader se encarga de cargar desde un archivo ubicado en el
 * sistema de archivos distribuido un stream que lee en un objeto Pcap,
 * para que sea procesado en mapreduce.
 *
 * @author Jorge Cifuentes <jorge.cifuentes95@gmail.com>,
    <jorge.cifuentesf@estudiante.uam.es>
 * @date 24 apr 2018
 */
class PcapFileRecordReader extends RecordReader<LongWritable, Pcap> {

    private Configuration conf;
    private FileSplit fileSplit;
    private boolean isProcessed = false;

    private LongWritable key = new LongWritable(1);
    private Pcap value;

    public void initialize(InputSplit inputSplit, TaskAttemptContext
        taskAttemptContext)
            throws IOException, InterruptedException {
        this.fileSplit = (FileSplit) inputSplit;
        this.conf = taskAttemptContext.getConfiguration();
    }

    @Override
    public LongWritable getCurrentKey() throws IOException,
        InterruptedException {
        return key;
    }

    @Override
    public Pcap getCurrentValue() throws IOException, InterruptedException {
        return value;
    }

    @Override
    public float getProgress() throws IOException, InterruptedException {
        return (isProcessed == true) ? 1.0f : 0.0f;
    }

    @Override
    public void close() throws IOException {
    }

    @Override
    public boolean nextKeyValue() throws IOException {
        if (!isProcessed) {
            Path file = fileSplit.getPath();
            FileSystem fs = file.getFileSystem(conf);

            FSDataInputStream in = null;
            in = fs.open(file);

            final Pcap pcap = Pcap.openStream(in);
            value = pcap;

            isProcessed = true;
            return isProcessed;
        }

        return false;
    }
}
```

Figura B.3: Código fuente de `tfg.Pcap.PcapFileRecordReader`

```java
/**
 * Esta clase ejecuta el mapeo. Inicializa los archivos de flujos
 * y llama a CustomPacketHandler para que itere sobre los paquetes.
 * Cierra el archivo de flujos llamando a saveFlows, y copia el
 * archivo de resultados temporal al sistema distribuido con
 * copyFromLocalFile. Devuelve la ruta del sistema distribuido al
 * Reducer.
 *
 * @author Jorge Cifuentes <jorge.cifuentes95@gmail.com>,
    <jorge.cifuentesf@estudiante.uam.es>
 * @date 26 apr 2018
 */
public class FileMapper extends Mapper<
        LongWritable, Pcap, // Recibe un par (long, Pcap) de parte del
            PcapFileInputFormat
        LongWritable, Text  // Lo mapea a un par (long, texto)
> {

    static {
        System.load("/home/jorgecf/libfp.so");
    }

    /**
     * Inicializa los archivos y demas variables que necesitaremos
     * en la libreria nativa para poder usarla.
     *
     * @param targetFolder Ruta de la carpeta donde guardar los archivos
     *  de salida temporales.
     */
    private native void initFlows(String targetFolder);

    /**
     * Pinta en un archivo temporal toda la informacion que ha ido
     * almacenando relativa a los flujos.
     *
     * @return String Ruta local del archivo temporal creado.
     */
    private native String saveFlows();

    @Override
    public void map(LongWritable key, Pcap value, Context context) throws
        IOException, InterruptedException {

        this.initFlows("/tmp");

        CustomPacketHandler ph = new CustomPacketHandler();
        value.loop(ph);

        String localTmp = this.saveFlows();
        FileSystem fs = FileSystem.get(context.getConfiguration());
        Path targetFile = new Path("/flows" + "/" +
            FilenameUtils.getName(localTmp));

        fs.mkdirs(targetFile.getParent()); // no falla si ya existe el
            directorio
        fs.copyFromLocalFile(new Path(localTmp), targetFile);

        context.write(key, new Text(targetFile.toString()));
    }
}
```

Figura B.4: Código fuente de `tfg.FileMapper`

```java
/**
 * Ejecuta la reduccion de los resultados de todos los FileMapper agregados.
 * Reduce todos los Flows de los achivos intermedios creados por los mapper en
 * uno solo, con los Flows iguales en diferentes archivos reducidos (agregados)
 * en sus campos.
 *
 * @author Jorge Cifuentes <jorge.cifuentes95@gmail.com>,
    <jorge.cifuentesf@estudiante.uam.es>
 * @date 26 apr 2018
 */
public class FileReducer extends Reducer<
        LongWritable, Text, // Recibe un par (long, lista{text})
        LongWritable, Text  // Reduce a par (long, texto)
> {

    @Override
    protected void reduce(LongWritable key, Iterable<Text> values, Context
        context)
            throws IOException, InterruptedException {

        FileSystem fs = FileSystem.get(context.getConfiguration());

        FlowsHandler fh = new FlowsHandler();

        FSDataInputStream inputStream;
        String partialFlows;

        /* Iteramos sobre cada archivo de Flows temporal */
        for (Text v : values) {
            inputStream = fs.open(new Path(v.toString()));
            partialFlows = IOUtils.toString(inputStream, "UTF-8"); // Leemos
                el archivo temporal desde hadoop

            /* Iteramos sobre cada linea de este archivo de Flows temporal (1
                linea = 1 Flow) */
            for (String flowLine : StringUtils.split(partialFlows, '\n')) {
                fh.add(StringUtils.split(flowLine, ',')); // Obtenemos los
                    atributos del Flow
            }

            inputStream.close();
        }

        context.write(key, new Text(fh.toString()));
    }

}
```

Figura B.5: Código fuente de `tfg.FileReducer`

```java
/**
 * Se encarga de hacer el "trabajo sucio" de parsear y actualizar todos
 * los campos relativos al Flow.
 *
 * @author Jorge Cifuentes <jorge.cifuentes95@gmail.com>,
    <jorge.cifuentesf@estudiante.uam.es>
 * @date 26 apr 2018
 */
public class Flow {

    private String ip_origin,  ip_destination,  mac_origin, mac_destination;
    private String protocol;
    private String port_origin, port_destination;

    private long n_incoming_packets, n_incoming_bytes;

    private long flow_starttime, flow_endtime;

    private long duration;
    private float pack_duration, byte_duration;

    private long max_pack_size, min_pack_size;
    public void reduceWith(Flow newFlow) {
        this.n_incoming_packets += newFlow.n_incoming_packets;
        this.n_incoming_bytes += newFlow.n_incoming_bytes;

        this.flow_starttime = Math.min(this.flow_starttime,
            newFlow.flow_starttime);
        this.flow_endtime = Math.max(this.flow_endtime, newFlow.flow_endtime);

        this.duration = this.flow_endtime - this.flow_starttime + 1;

        this.pack_duration = this.n_incoming_packets / this.duration;
        this.byte_duration = this.n_incoming_bytes / this.duration;

        this.max_pack_size = Math.max(this.max_pack_size,
            newFlow.max_pack_size);
        this.min_pack_size = Math.min(this.min_pack_size,
            newFlow.min_pack_size);
        this.avg_pack_size = this.n_incoming_bytes / this.n_incoming_packets;
    }

    @Override
    public int hashCode() {
        return this.ip_origin.hashCode() + this.ip_destination.hashCode() +
            this.port_origin.hashCode()
                + this.port_destination.hashCode() + this.protocol.hashCode();
    }

    @Override
    public boolean equals(Object other) {
        if (other == this) return true;
        if (other == null || !(other instanceof Flow)) return false;
        return (this.ip_origin.equals(((Flow) other).ip_origin)
                && this.ip_destination.equals(((Flow) other).ip_destination)
                && this.port_origin.equals(((Flow) other).port_origin)
                && this.port_destination.equals(((Flow)
                    other).port_destination)
                && this.protocol.equals(((Flow) other).protocol));
    }
}
```

Figura B.6: Código fuente de `tfg.Flows.Flow`

```java
/**
 * Se encarga de manejar la lista de Flows. Agrega los nuevos y
 * actualiza (reduce) los ya existentes.
 *
 * @author Jorge Cifuentes <jorge.cifuentes95@gmail.com>,
    <jorge.cifuentesf@estudiante.uam.es>
 * @date 26 apr 2018
 */
public class FlowsHandler {

    private Map<Integer, Flow> flows;

    public FlowsHandler() {
        this.flows = new HashMap<Integer, Flow>();
    }

    /**
     * Agrega un nuevo Flow, o lo reduce si ya existe.
     *
     * @param fields Lista de campos que conforman el nuevo Flow.
     */
    public void add(String[] fields) {
        Flow newFlow = new Flow(fields);
        Flow existingFlow = this.flows.get(newFlow.hashCode());

        if (existingFlow != null) {
            existingFlow.reduceWith(newFlow);
        } else {
            this.flows.put(newFlow.hashCode(), newFlow);
        }

    }

    public int getFlowsCount() {
        return this.flows.size();
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();

        Iterator it = this.flows.keySet().iterator();
        while (it.hasNext()) {
            sb.append(this.flows.get(it.next()).toString());
            it.remove();
        }

        return sb.toString();
    }
}
```

Figura B.7: Código fuente de `tfg.Flows.FlowsHandler`

```java
/**
 * Esta clase pinta en un archivo los valores de los
 * contadores del Apache Hadoop Job que se le pasa.
 *
 * @author Jorge Cifuentes <jorge.cifuentes95@gmail.com>,
     <jorge.cifuentesf@estudiante.uam.es>
 * @date 23 apr 2018
 */
public class JobResultsWriter {

    private static final String LINE_SEPARATOR = "\n";

    private CSVPrinter printer = null;
    private CSVFormat format;
    private String outputFile;

    private Job job;
    private long executionTime;

    private List<String> header = new ArrayList<String>();

    private enum FSCounter { // Contadores del File System
        FILE_BYTES_READ, FILE_BYTES_WRITTEN, HDFS_BYTES_READ,
            HDFS_BYTES_WRITTEN,
    }

    public JobResultsWriter(String outputFile, Job job, long executionTime)
        throws IOException {
        this.job = job;
        this.executionTime = executionTime;
        this.outputFile = outputFile;
    }

    public void write() throws IOException, InterruptedException {
        List<String> fields = new ArrayList<String>();

        /* Tiempo de ejecucion */
        fields.add(String.valueOf(this.executionTime));
        this.header.add("EXECUTION_TIME");

        Counters counters = this.job.getCounters();

        /* Contadores del File System */
        for (FSCounter c : FSCounter.values()) {
            long v = counters.findCounter("FileSystemCounters",
                c.name()).getValue();
            fields.add(String.valueOf(v));
            this.header.add(c.name());
        }

        /* Contadores del Job */
        for (JobCounter c : JobCounter.values()) {
            long v = counters.findCounter(c).getValue();
            fields.add(String.valueOf(v));
            this.header.add(c.name());
        }

        /* Contadores del Map-Reduce */
        for (TaskCounter c : TaskCounter.values()) {
            long v = counters.findCounter(c).getValue();
            fields.add(String.valueOf(v));
            this.header.add(c.name());
        }
```

```
60          /* Configuracion de la salida */
61          if (new File(this.outputFile).exists()) {
62              this.format =
                    CSVFormat.DEFAULT.withRecordSeparator(LINE_SEPARATOR); // no
                    queremos escribir multiples veces el header
63          } else {
64              this.format = CSVFormat.DEFAULT.withRecordSeparator(LINE_SEPARATOR)
65                      .withHeader(this.header.toArray(new String[0]));
66          }
67
68          /* Lo escribimos */
69          FileWriter fw = new FileWriter(outputFile, true);
70          this.printer = new CSVPrinter(fw, format);
71
72          this.printer.printRecord(fields);
73          this.printer.close();
74      }
75  }
```

Figura B.8: Código fuente de `tfg.JobResultsWriter`

```
1   /**
2    * Este PacketHandler se encarga de procesar cada paquete PCcapPacket
3    * llamando a la libreria nativa flow_process.
4    *
5    * @author Jorge Cifuentes <jorge.cifuentes95@gmail.com>,
         <jorge.cifuentesf@estudiante.uam.es>
6    * @date 26 apr 2018
7    */
8   public class CustomPacketHandler implements PacketHandler {
9
10    static {
11      System.load("/home/jorgecf/libfp.so");
12    }
13
14    /**
15     * Esta funcion es la principal, y se encarga de procesar el paquete,
16     * analizando sus campos y agregandolo al flujo que le corresponda.
17     *
18     * @param packet Representacion en bytes del paquete de red.
19     *
20     * @param sec Segundos del timeval de pcap_pkthdr en C.
21     * @param usec uSegundos del timeval de pcap_pkthdr en C.
22     * @param len Longitud del paquete de pcap_pkthdr en C.
23     * @param caplen Longitud capturada de pcap_pkthdr en C.
24     */
25    private native void processPacket(byte[] packet, long sec, long usec, long
          len, long caplen);
26
27    public boolean nextPacket(final Packet packet) throws IOException {
28
29      String usec_s = String.format("%06d", packet.getArrivalTime() % 1000000);
30      String sec_s = String.valueOf(packet.getArrivalTime()).replace(usec_s, "");
31
32      /* campos del pcap_pkthdr de C */
33      this.processPacket(packet.getPayload().getArray(), Long.parseLong(sec_s),
            Long.parseLong(usec_s),
34          ((PCapPacket) packet).getTotalLength(), ((PCapPacket)
                packet).getCapturedLength());
35
36      return true;
37    }
38  }
```

Figura B.9: Código fuente de `tfg.Pcap.CustomPacketHandler`

```
1  JNIEXPORT void JNICALL Java_tfg_Pcap_CustomPacketHandler_processPacket(JNIEnv
       *env, jobject thisObj, jbyteArray pcap, jlong sec, jlong usec, jlong len,
       jlong caplen) {
2
3    /* Parseo de array de jbytes a array de uint8_t */
4    jbyte  *jniData;
5    jniData = (*env)->GetByteArrayElements(env,pcap,0);
6
7    uint8_t *bp = (uint8_t*) jniData;
8
9    /* Parseo a timeval */
10   time_t s = (time_t) sec;
11   suseconds_t us = (suseconds_t) usec;
12
13   struct pcap_pkthdr h;
14   h.ts.tv_sec = s;
15   h.ts.tv_usec = us;
16   h.len = len;
17   h.caplen = caplen;
18
19   /* Procesamos el paquete */
20   process_packet(bp, h, NULL , 0);
21
22   (*env)->ReleaseByteArrayElements(env, pcap, jniData, JNI_ABORT);
23 }
24
25 JNIEXPORT void JNICALL Java_tfg_FileMapper_initFlows(JNIEnv *env, jobject
       thisObj, jstring targetFolder) {
26   uint8_t output_format = FP_STANDARD_OUTPUT;
27
28   /* Tiempo maximo entre un paquete de un flujo y el siguiente. */
29   expiration_flow_time = 100000 * 1000000L;
30
31   allocIPSessionPool(); /* Inicializaciones de la tabla de sesiones */
32     allocNodelPool();
33
34     /* Carpeta temporal donde almacenar los ficheros de salida */
35     const char *target = (*env)->GetStringUTFChars(env, targetFolder, 0);
36
37   export = export_to_file;
38
39   struct pcap_pkthdr h; /* Abrimos los archivos que vayamos a usar */
40   if(new_pcap_file(h.ts.tv_sec, target) != OK){ exit(ERROR); }
41
42     /* Dejamos libres los recursos de jni */
43   (*env)->ReleaseStringUTFChars(env, targetFolder, target);
44 }
45
46 JNIEXPORT jstring JNICALL Java_tfg_FileMapper_saveFlows(JNIEnv *env, jobject
       thisObj) {
47   last_packet_timestamp = INFINITO;
48   cleanup_flows();
49   char* retfile = close_flows();
50
51   jstring ret = (*env)->NewStringUTF(env, retfile);
52   return ret;
53 }
```

Figura B.10: Código con el paso de parámetros de Java a C con JNI